

CTOS

Operating System
Concepts Manual
Volume 1

UNISYS

UNISYS

CTOS[®]
Operating System
Concepts
Manual
Volume 1

Copyright © 1991, 1992 Unisys Corporation
All Rights Reserved
Unisys is a trademark of Unisys Corporation

CTOS III 1.0
CTOS II 3.4
CTOS I 3.4
CTOS/XE 3.4
Development Utilities 12.2
Standard Software 12.2
Video Access Method 4.0

July 1992

Printed in USA
4360 1186-000

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information or software material, including direct, indirect, special or consequential damages.

You should be careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED RIGHTS LEGEND. Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.227-7013 and FAR 52.227-14 for commercial computer software.

Convergent, Convergent Technologies, CTOS, and NGEN are registered trademarks of Convergent Technologies, Inc.

OFIS is a registered trademark of Unisys Corporation.

BTOS is a trademark of Unisys Corporation.

Context Manager, Context Manager/VM, CT-Net, CTOS/VM, Document Designer, Generic Print System, PC Emulator, Print Manager, Shared Resource Processor, Solution Designer, SRP, Voice/Data Services, Voice Processor, and X-Bus are trademarks of Convergent Technologies, Inc.

OS/2 is a registered trademark of International Business Machines Corporation. Intel is a registered trademark of Intel Corporation. Lotus is a registered trademark of Lotus Development Corporation.



Product Information Announcement

● New Release ○ Revision ○ Update ○ New Mail Code

Title

CTOS® Operating System Concepts Manual, Volume 1

This Product Information Announcement announces the release and availability of the CTOS Operating System Concepts Manual.

This manual describes the CTOS III 1.0, CTOS II 3.4, and CTOS I 3.4 workstation operating systems and the CTOS/XE 3.4 operating system for shared resource processors. The manual explains how CTOS works and provides an orientation to the basic concepts the CTOS programmer needs to know. New topics include demand paging, dynamic link libraries, semaphores, DMA buffer management, name management, and run file resource management. In addition, the manual provides an overview of the memory management styles for multipartition (CTOS I), variable partition (CTOS II), and demand-paged, virtual memory (CTOS III) operating systems.

Please address all technical communications relative to this document to UNISYS, Multimedia Publishing M/S 18-007, 2700 North First Street, San Jose, CA 95134-2028.

To order additional copies of this document

- United States customers call Unisys Direct at 1-800-448-1424.
- All other customers, contact your Unisys Sales Office.
- Unisys personnel use the Electronic Literature Ordering (ELO) system.

Contact your Unisys representative for pricing information.

4360 1186-000 Volume 1 only

4360 2630-000 Volume 2 only

4357 9507-100 Volumes 1 and 2 with binders and slipcases

CTOS is a registered trademark of Convergent Technologies, Inc.

Announcement only:

SA, SN, SU, SW, PR5

Announcement and attachments:

System: CTOS

Release:

CTOS III 1.0

CTOS II 3.4

CTOS I 3.4

CTOS/XE 3.4

Development Utilities 12.2

Standard Software 12.2

Video Access Method 4.0

Part Number: 4360 1186-000

Page Status

Page	Issue
v through xxxvi	Original
xxxvii through xlviii	Original
1-1 through 1-8	Original
2-1 through 2-24	Original
3-1 through 3-15	Original
3-16	Blank
4-1 through 4-15	Original
4-16	Blank
5-1 through 5-8	Original
6-1 through 6-10	Original
7-1 through 7-4	Original
8-1 through 8-16	Original
9-1 through 9-5	Original
9-6	Blank
10-1 through 10-23	Original
10-24	Blank
11-1 through 11-59	Original
11-60	Blank
12-1 through 12-49	Original
12-50	Blank
13-1 through 13-5	Original
13-6	Blank
14-1 through 14-2	Original
15-1 through 15-8	Original
16-1 through 16-6	Original
17-1 through 17-3	Original
17-4	Blank
18-1 through 18-29	Original
18-30	Blank
19-1	Original
19-2	Blank
20-1 through 20-8	Original
21-1	Original
21-2	Blank

Page	Issue
22-1 through 22-3	Original
22-4	Blank
23-1 through 23-5	Original
23-6	Blank
24-1 through 24-17	Original
24-18	Blank
25-1 through 25-14	Original
26-1 through 26-43	Original
26-44	Blank
27-1 through 27-13	Original
27-14	Blank
Index-1 through 41	Original
Index-42	Blank

Contents

About This Manual	xxxvii
--------------------------------	--------

Section 1. Introduction to CTOS

What Is CTOS?	1-1
CTOS Foundation	1-2
Event-Driven, Priority-Ordered Process Scheduling	1-3
Message-Based Operation	1-3
Multiprogramming	1-3
Multitasking (Multithreading)	1-4
Nationalization	1-4
Networking Built-In	1-4
Protected Mode Enhancements	1-4
Caching	1-5
Extended Native Language Support (ENLS)	1-5
Multiple and International Keyboard Support	1-5
Protected Mode Operation	1-6
Real Mode Application Support	1-6
SCSI Management	1-6
Variable Partitions With Code Sharing Capability	1-7
CTOS III Enhancements	1-7
Demand Paging	1-7
Dynamic Link Libraries (DLLs)	1-7
Name Management	1-8
Semaphores	1-8

Section 2. Overview of Operating System Concepts

What is a Process?	2-1
What Constitutes the Kernel?	2-1
Event-Driven Priority-Ordered Scheduling	2-2
Interprocess Communication (IPC)	2-2
System Service Processes	2-3
System Service Filters	2-4
Inter-CPU Communication (ICC)	2-4

Contents

Command Interpreter: the Executive	2-4
File Management System	2-5
User-Written Device Handlers	2-5
Distributed Environment and Clustering	2-6
Local Resource-Sharing Networks (Clusters)	2-6
Network	2-6
Operating System Types	2-7
Workstation Operating Systems	2-8
Shared Resource Processor Systems	2-9
Programs and Run Files	2-10
Setting Up the Operating Environment	2-10
Partitions and User Numbers	2-11
Memory Management Styles	2-11
Multipartition Memory Management	2-12
Variable Partition Memory Management	2-12
Virtual Memory Management	2-13
Memory Organization at System Initialization	2-14
Variable Partition and Multipartition	
Operating Systems	2-14
Virtual Memory Operating Systems	2-16
Loading Applications	2-18
Bringing Applications into Memory	2-18
Managing Physical Memory	2-19
Swapping	2-19
Demand Paging	2-20
Virtual Code Management	2-22
Application Partition Memory Organization	2-22
Dynamic Linking	2-24
Code and Data Sharing	2-24

Section 3. Demand Paging

Demand Paging Overview	3-1
Demand Paging Terminology	3-2
Demand Paging: A Memory Management Style	3-3
Benefits of Demand Paging	3-4
Page Mapping	3-5
Allocating Linear Address Space	3-6
Oversubscribing the Physical Address Space	3-8
Replacing Pages	3-9

Cleaning Pages	3-11
Prefaulting Pages	3-11
Page Locking	3-12
Paging Service Components	3-12
Executing Real Mode Applications	3-13
Querying Paging Statistics	3-13
Tuning Paging Through System Configuration	3-14
Demand Paging Operations	3-15

Section 4. Using CTOS Operations

Introduction to Using CTOS Operations	4-1
Before You Begin Programming on CTOS	4-1
Naming Conventions	4-2
Programmatic Interface	4-2
Example CTOS Call in C Language	4-3
Operation Types	4-4
Object Module Procedures	4-5
System-Common Procedures	4-5
Kernel Primitives	4-6
Using the Request Procedural Interface	4-6
Using the Kernel Primitives	4-7
I/O Interface Levels	4-8
Addressing Memory	4-10
Logical Memory Address	4-12
Linear Memory Address	4-13
Physical Memory Address	4-13
Advantages to Protected Mode Memory Addressing	4-14
Extended Memory	4-14
Protection	4-14
Selecting Operations for Program Portability	4-15

Section 5. Program Management

What Is Program Management?	5-1
What Is a Program?	5-1
Linking a Program	5-3
Loading a Program Into Memory	5-4

Program Termination	5-5
Loading the Exit Run File	5-5
Notifying Other Programs of Termination	5-5
Deallocating System Resources	5-6
Error Handling	5-6
Program Management Operations	5-7
Error Handling	5-7
Normal Program Exit	5-8
 Section 6. Parameter Management	
What is Parameter Management?	6-1
Program Using Parameter Management	6-1
Parameters	6-2
Structures and Operations Overview	6-3
Application System Control Block (ASCB)	6-3
Variable Length Parameter Block (VLPB)	6-4
Querying Parameters In the VLPB	6-4
Example of a VLPB for the Rename Command	6-6
Operations for Constructing the VLPB	6-8
VLPB Structure	6-9
Parameter Management Operations	6-10
Constructing Parameters	6-10
Querying Parameters	6-10
 Section 7. Input/Output	
In This Section	7-1
Manual Sections Describing I/O	7-1
Device Independence Versus Dependence	7-3
I/O Facilities	7-3
 Section 8. Sequential Access Method	
What is the Sequential Access Method?	8-1
Customizing the Sequential Access Method	8-2
Byte Stream	8-3
Using a Byte Stream	8-3
Types of Byte Streams	8-4
Disk Byte Streams	8-5
Printer Byte Streams	8-5
Generic Print System Byte Streams	8-6
Pre-GPS Spooler Byte Streams	8-7

Keyboard Byte Streams	8-8
Communications Byte Streams	8-8
X.25 Byte Streams	8-9
Video Byte Streams	8-9
Sequential Access Byte Streams	8-9
Device/File Specifications	8-10
Device/File Specification Parsing	8-14
SAM Operations	8-15
Basic	8-15
Advanced	8-16

Section 9. Device Dependent SAM

What is Device-Dependent SAM?	9-1
Mapping to Device-Dependent Operations	9-1
Device-Specific Operations	9-2
Device-Dependent SAM Operations	9-4

Section 10. Video

What is the Video Facility?	10-1
Character-Map and Bit-Map Video	10-1
Video Attributes	10-2
Video Software	10-2
Using the SAM Operations	10-3
Using the Current Screen Setup	10-3
Using SAM Directly	10-4
Augmenting the SAM Operations	10-4
Special Characters in Video Byte Streams	10-5
Multibyte Escape Sequences	10-5
Controlling Screen Attributes	10-5
Controlling Character Attributes	10-6
Controlling Scrolling and Cursor Positioning	10-6
Dynamically Redirecting a Video Byte Stream	10-6
Automatically Pausing Between Full Frames	10-7
Miscellaneous Functions	10-7
Using QueryVidBs	10-8
Using the Video Access Method (VAM)	10-8
Using Video Display Management (VDM)	10-8
Reinitializing the Video Subsystem	10-9
Forms-Oriented Interaction	10-10
Advanced Text Processing	10-11

Workstation Video Capabilities	10-11
Character Cell	10-14
Character-map	10-15
Video Attributes	10-15
Font	10-15
Cursor	10-16
Video Refresh	10-16
Writing Portable Video Programs	10-16
Video Data Structures	10-17
Programming Using Color	10-17
Video Operations	10-18
VAM Operations	10-18
Video Requests	10-20
VDM Operations	10-20
Color Programming Operations	10-22
Direct Access to Video Data Structures	10-23

Section 11. Keyboard and I-Bus Management

What is Keyboard Management?	11-1
Keyboard Terminology	11-2
Keyboard Management Features	11-5
Keyboard Management Overview	11-7
Preprocessing	11-9
Postprocessing	11-10
Keyboard Modes	11-10
Unencoded Mode	11-11
Character mode	11-12
Other Keyboard Modes	11-12
Comparing the Keyboard Modes	11-13
Keyboard Hardware Protocols	11-13
Reading the Keyboard	11-14
Using ReadKbdInfo	11-14
Information Returned to the Requesting Application ..	11-15
Status Word Information	11-15
Type-Ahead Buffer	11-16
Writing to the Type-Ahead Buffer	11-17
Keyboard Data Block Organization	11-18
Finding a Key Definition	11-20
Translation Data Block Organization	11-20

Translating	11-22
Supporting Tables	11-22
Selecting the Translation Table to Use	11-23
Obtaining Other Keyboard Details	11-25
Contents of the Default K1 Translation Data Block	11-25
Decoding	11-26
Comparing Data Blocks	11-26
Emulating	11-28
Emulation Examples	11-28
Redefining Keys	11-30
Emulating LEDs	11-31
Customizing the Keyboard Data Blocks	11-31
Customizing the System Keyboard Data Blocks	11-32
Building a New System Keyboard File	11-32
Customizing Application-Specific Data Blocks	11-32
Dynamically Customizing Data Blocks for Application Use	11-33
If Your Application Uses NLS Keyboard Tables	11-33
Modifying Data Blocks	11-34
Posting Data Blocks	11-36
Reading Keyboard Data Blocks	11-36
Using ReadOsKbdTable	11-37
Using Byte Streams	11-37
Binary Data Block File to Local Memory	11-38
Data Block in an Object Module	11-39
Comparing Methods of Reading Data Blocks	11-39
Application Customization Examples	11-40
Multibyte Strings	11-40
Diacritics	11-40
Chords that Toggle	11-40
System Profile Keyboard	11-41
Matching Keyboards to Data Blocks	11-42
Application Profile Keyboard	11-43
Multiple Keyboard Profiles	11-43
Mapping Keyboard IDs	11-44
Setting Keyboard Options	11-45
Sticky Keys	11-45
Character Repeating	11-45

Contents

System Input Process	11-46
Playback Mode	11-47
Recording Mode	11-49
Submit File Escape Sequences	11-49
Read-Direct Escape Sequences	11-51
Action Key	11-52
At Program Termination	11-54
Keyboard and Video Independence	11-54
I-Bus Device Management	11-54
Keyboard and I-Bus Management Operations	11-56
Basic	11-56
Advanced	11-57
I-BUS Management	11-59
Magnetic Card Reader	11-59

Section 12. File Management

What is File Management?	12-1
Overview of File System Capabilities	12-1
Efficiency	12-1
Reliability	12-2
Convenience	12-3
Structured File Access Methods	12-3
Access to Local Files	12-4
File Specifications	12-4
Node	12-5
Volume	12-5
Directory	12-7
File	12-7
Password	12-8
Directory and File Specifications	12-9
Abbreviated Specifications	12-10
Automatic Volume Recognition	12-11
File Protection	12-11
Protection by Password	12-12
Volume Password	12-12
Directory Password	12-12
File Password	12-13
Device Password	12-13

Using a Password For Access	12-13
Protection by Protection Level	12-14
How Protection Levels Work	12-14
How The Operating System Validates	
Protection Levels	12-16
Protection by Volume Encryption	12-18
Creating and Accessing a File	12-20
Structured File Access Methods	12-20
Byte Streams	12-20
File Management Operations	12-20
Logical File Address	12-21
File Handle	12-22
From Creating to Deleting a File	12-22
Creating a File	12-22
Opening a File	12-24
Reading and Writing a File	12-25
Closing a File	12-26
Deleting a File	12-27
Local File System	12-28
LfsToMaster	12-29
Volume Control Structures	12-29
Volume Home Block	12-30
Allocation Bit Map and Bad Sector File	12-32
File Header Block	12-32
Disk Extent	12-32
Extension File Header Block	12-32
Master File Directory and Directories	12-33
System Directory	12-33
System Data Structures	12-34
User Control Block	12-34
Device Control Block	12-35
Building and Parsing File Specifications	12-35
Terms	12-36
Using The File Building and Parsing Operations	12-36
Layers of Access	12-37
Validating File Specifications	12-38
Syntax Errors	12-40
Wild Card Operations	12-41
\$ Directory	12-42

File Management Operations	12-43
Basic	12-43
Basic Utility Operations	12-43
File Attributes	12-44
Default Path	12-45
Directories	12-45
Long-Lived Files	12-46
File Handle Operations	12-46
Parsing Specifications	12-47
Asynchronous File I/O	12-49
Volume Data Structures	12-49
 Section 13. Disk Management	
What is Disk Management?	13-1
Accessing a Disk Device	13-1
Device Specification and Password	13-2
Memory Disk	13-3
Cached Memory Disk	13-3
Disk Partitions	13-4
Disk Management Operations	13-5
 Section 14. Printing Management	
Generic Print System Components	14-1
Interface Considerations	14-2
 Section 15. Communications Programming	
What is Communications Programming?	15-1
What SamC Is Used For	15-1
What Programs Use SamC	15-2
What Programs Cannot Use SamC	15-2
At the Device-Independent Interface Level	15-2
At the Device-Dependent Interface Level	15-3
Using the SamC Operations	15-4
Asynchronous Interface	15-4
The AcquireByteStreamC Operation (Low-Level Open)	15-5
Dynamically Changing Parameters	15-5
Querying and Setting Status Lines	15-5
The CheckForOperatorRestartC Operation	15-5
SamC Operations	15-6

Section 16. Serial Port Management

Access Below the Byte Stream Level (CommLine)	16-1
Serial Port Operations	16-2
Using InitCommLine	16-2
Using ResetCommLine	16-3
Using ChangeCommLineBaudRate	16-3
Using ReadCommLineStatus	16-4
Using WriteCommLineStatus	16-4
Serial Port Management Operations	16-5

Section 17. SRP Terminal Management

Program Access to Ports	17-1
SRP Terminal Management Operations	17-3

Section 18. SCSI Device Management

What is SCSI Management?	18-1
SCSI Management Terminology	18-3
Overview of the SCSI Manager's Capabilities	18-4
Convenience	18-4
Efficiency	18-5
Reliability	18-5
Creating a SCSI Path	18-5
Configuring a SCSI Manager	18-7
Defining a Unique Path	18-7
Specifying Path Parameters	18-8
Changing and Querying Path Parameters	18-9
Using the Path Handle	18-9
SCSI Access Modes	18-10
SCSI Passwords	18-11
File System and the SCSI Manager	18-12
SCSI Command Structure	18-14
Error Conditions and SCSI Sense Data	18-16
SCSI Manager Target Mode	18-20
Automatic Target Mode Functions	18-21
Explicitly Enabling Target Mode Functions	18-21
Managing Data Packets	18-22
Operating Methods for Target Mode Commands	18-22
Using a Target Check or Wait Operation	18-23

Contents

Example of Using the SCSI Manager	18-23
SCSI Management Operations	18-26
SCSI Paths	18-26
Basic SCSI I/O	18-27
Advanced SCSI	18-27
Target Mode	18-28
 Section 19. Generic Print Access Method	
What Is the Generic Print Access Method?	19-1
 Section 20. Structured File Access Methods	
What is Structured File Access?	20-1
ISAM	20-2
RSAM	20-3
DAM	20-4
Hybrid Access Patterns	20-4
Modifying and Reading Data Files	20-5
Selecting a File Access Method	20-6
Disk Space	20-6
Access Time	20-7
Structured File Access Methods Operations	20-8
 Section 21. Indexed Sequential Access Method	
What is ISAM?	21-1
 Section 22. Record Sequential Access Method	
What is RSAM?	22-1
RSAM Files and Records	22-1
RSAM Working Area	22-2
RSAM Buffer	22-2
RSAM Operations	22-3
Basic	22-3
Advanced	22-3

Section 23. Direct Access Method

What is DAM?	23-1
DAM Files, Records, and Record Fragments	23-1
DAM Working Area	23-2
DAM Buffer	23-2
Buffer Size and Sequential Access	23-3
Buffer Management Modes	23-3
DAM Operations	23-4
Basic	23-4
Advanced	23-4

Section 24. Memory Management

What is Memory Management?	24-1
Memory Management Terminology	24-1
Partition Memory	24-2
Global Linear Address Space	24-3
Segments	24-3
Addressing a Byte in a Segment	24-3
Segment Limit and Huge Segments	24-4
Code, Static Data, and Dynamic Data Segments	24-4
Expand Up and Expand Down Segments	24-5
From Source Modules to Program in Memory	24-6
Models of Segmentation	24-6
Run Files	24-6
Application Partition Memory Organization	24-8
Allocating and Deallocating Partition Memory	24-9
Order of Deallocating Partition Memory	24-10
Using Long-Lived Memory	24-10
Using Short-Lived Memory	24-11
Using Global Linear Address Space	24-11
Obtaining Memory Addressed by a Single Selector	24-12
Obtaining Memory Addressed by a Selector Sequence	24-12
Memory Management Operations	24-13
Short-Lived Memory	24-13
Long-Lived Memory	24-14
Short-Lived and Long-Lived Memory	24-14
I/O Management	24-15
Selector Management	24-15
Global Linear Address Space Management	24-17

Section 25. Cacher

What is the Cacher?	25-1
Cacher Terminology	25-2
Why Use the Cacher?	25-3
Cacher Data Structures	25-4
Cache Pool Descriptor	25-4
Cache Entry Descriptor	25-5
Initializing a Cache	25-5
Allocating Cache Memory	25-5
Calculating Number of Cache Entries	25-6
Formatting Cache Memory	25-6
Hashing	25-8
Obtaining a Cache Entry	25-9
Cache Optimizations	25-10
Chaining Cache Buffers	25-10
Deferring a Cache Hit	25-10
Preventing Buffer Stealing	25-10
Releasing a Cache Entry	25-11
Conditions Before Release of an Entry	25-11
At Release of an Entry	25-12
Flushing Cache Entries	25-12
Obtaining Cache Status	25-12
Obtaining Cache Statistics	25-13
Closing a Cache Pool	25-13
Cacher Operations	25-14

Section 26. Utility Operations

What are Utility operations?	26-1
Accessing Resources in Disk Files	26-2
What Are Resources?	26-2
Why Resource Management?	26-3
Resource Work Area	26-4
Stepping On Files	26-5
Process Overview	26-5
Starting And Ending a Resource Session	26-7
Initializing Resource Access	26-8
Saving to Disk	26-9
Copying Resources	26-11
Copying One Resource at a Time	26-11
Using a Base RWA	26-11
Copying Multiple Resources	26-12

Copying a Resource From a CTOS Data File	26-13
Deleting Resources	26-13
Copying The Nonresource Portion of a Run File	26-14
Accessing Resources in Memory	26-14
Comparing Strings	26-14
Using StringsEqual and NlsULCmpB	26-15
Using WildCardMatch	26-15
Handling Nationalized Strings	26-15
Customizing the User Interface	26-16
Directing Data to a Byte Stream	26-16
Handling Commands	26-18
Managing Names	26-18
Classes	26-18
Configuring the Name Heap	26-19
Using the Name Management Operations	26-19
Manipulating Error Messages	26-20
Obtaining the System Date and Time	26-21
Parsing Configuration Files	26-23
Parsing User Configuration Files	26-23
Parsing Standard Configuration Files	26-24
Parsing Nonstandard Configuration Files	26-25
Using Unique Workstation Hardware IDs	26-27
Performing Miscellaneous Tasks	26-27
Building a Single-Line Text Editor	26-27
Comparing Logical Addresses	26-28
Copying Files	26-28
Determining Monitor Resolution	26-28
Writing Records to the System Log File	26-28
Informing User of Waiting Mail	26-28
Utility Operations	26-29
Accessing Resources in Disk Files	26-29
Accessing Resources in Memory	26-30
Comparing Strings	26-31
Customizing the User Interface	26-32
Directing Data to a Byte Stream	26-33
Handling Commands	26-35
Managing Names	26-35
Manipulating Error Messages	26-35
Obtaining the System Date and Time	26-36
Obtaining Versions of System Services	26-38
Parsing Configuration Files	26-40
Using Workstation Hardware IDs	26-42
Performing Miscellaneous Tasks	26-43

Section 27. System Definitions

System Definitions in This Section	27-1
Methods of Obtaining System Information	27-9
System Definition Operations	27-10
Cluster Management	27-10
Disk Management	27-10
File Management	27-10
Operating System	27-10
User Name Management	27-13
Video	27-13

Section 28. Multiprogramming

Benefits of Multiprogramming	28-1
Simple Multiprogramming Event Sequence	28-1
Multiprogramming Subjects	28-2

Section 29. Process Management

How is a Process Perceived?	29-1
By an End User	29-1
By a Programmer	29-1
By the Operating System	29-2
Process Management	29-2
Context of a Process	29-2
Process Priorities and Scheduling	29-3
Process States	29-4
Process Management Operations	29-7

Section 30. Interprocess Communication

What is Interprocess Communication?	30-1
An IPC Example	30-1
What Really Happens	30-2
Request Procedural Interface	30-2
System Service	30-3
IPC Summary	30-3
Other IPC Applications	30-4
Communication Within an Application Partition	30-4
Communication Between Application Partitions	30-5
Synchronization	30-6
Resource Management	30-7

Why Understand IPC?	30-9
Request Codes	30-9
Interprocess Communication (IPC) Components	30-10
Kernel Primitives for Sending a Message	30-12
Request and Respond	30-12
Send	30-15
ForwardRequest and RequestDirect	30-15
Kernel Primitives for Receiving a Message	30-16
Wait	30-16
Check	30-17
The Exchange	30-17
Types of Exchanges	30-18
Exchange Allocation	30-18
Sending a Message to an Exchange	30-19
Waiting for a Message at an Exchange	30-21
Exchange Queues	30-22
The Message	30-23
Request Block Format	30-24
Standard Request Block Header	30-25
Control Information	30-26
Routing Code	30-27
Request Data Item	30-27
Response Data Item	30-27
Example Request Block	30-28
Accessing System Services	30-30
Using the Request Procedural Interface	30-30
Using the Kernel Primitives Directly	30-31
Cluster/Network Communication	30-32
Cluster Configuration	30-33
Workstation Agent	30-33
Master Agent	30-33
Extending Resource Sharing	30-34
Routing by Handle	30-34
Routing by Specification	30-41
Rules for Routing by Specification	30-41
Expanding Specifications	30-42
Routing Code	30-43
Routing Requests	30-45
Network Routing	30-46
Shared Resource Processor Routing	30-47
Exchange Routing	30-49

Filter Process	30-51
Interprocess Communication Summary	30-52
Interprocess Communication Operations	30-54

Section 31. Semaphores

What is a Semaphore?	31-1
Who Uses Semaphores?	31-2
Semaphore Terminology	31-2
Semaphore Types	31-4
System and RAM Semaphores	31-4
Mutual Exclusion Semaphores	31-5
Critical Section Semaphores	31-5
Noncritical Semaphores	31-6
Signaling Semaphores	31-6
Opening a System Semaphore	31-6
Providing Exclusive Access to a Resource	31-7
Noncritical Semaphores	31-7
Locking a Noncritical Semaphore	31-7
Clearing a Noncritical Semaphore	31-8
Critical Section Semaphores	31-8
Using the Critical Section Operations	31-9
Terminating and Suspending Process	31-10
Mutual Exclusion on CTOS	31-10
Behind-the-Scenes Use of a Semaphore	31-11
Implementing a Simple Mutual Exclusion Semaphore ..	31-11
Signaling and Synchronizing Processes	31-12
Using the Signaling Semaphore Operations	31-12
The Producer/Consumer Model	31-13
CTOS Primitive Solutions to the	
Producer/Consumer Model	31-15
Using RAM Semaphores	31-16
RAM Semaphore Variable	31-17
Guidelines to Using RAM Semaphores	31-18
Semaphore Operations	31-20
System Semaphores	31-20
Mutual Exclusion	31-20
Critical Section	31-20
Signaling and Synchronization	31-20

Section 32. Inter-CPU Communication

What is Inter-CPU Communication?	32-1
ICC Terminology	32-2
Slot Numbers	32-2
Shared Resource Processor Routing Types	32-3
Blocks	32-5
CPU Description Table	32-6
ICC Segment	32-6
Ring Queues of Block Type Buffers	32-6
Bus Adresses	32-6
Ring Queues of Requests and Responses	32-7
Buffer Ownership Table	32-7
Buffer Wait Tables	32-7
Doorbell Interrupt	32-7
Interboard Routing	32-8
Sending a Request	32-8
Receiving a Request	32-11
Sending a Response	32-11
Receiving a Response	32-12
Sending and Receiving Messages	32-12
Mediating Buffer Wait Conditions	32-14
ICC Operations	32-18

Section 33. System Services Management

What is System Services Management?	33-1
System Service Terminology	33-1
Overview of Operation	33-2
Built-In System Services	33-5
Dynamically Installable System Services	33-6
Request Routing Table	33-6
System Service Package	33-7
Requests	33-7
The System Service	33-8
Guidelines for Writing a System Service	33-8
Initialization and Conversion to a System Service	33-8
System Service Main Program	33-13
Restrictions and Requirements of Operation	33-14
Guidelines for Defining Requests	33-14
Creating Loadable Request Files	33-16

System Requests	33-17
Termination and Abort Requests	33-18
Termination Request to the File System	33-19
Swapping Requests	33-19
Change User Number Requests	33-20
Converting to a Multi-Instance Service	33-21
Filters	33-22
Types of Filters	33-22
Replacement	33-22
One-Way Pass Through	33-22
Two-Way Pass Through	33-24
System Requests for Filters	33-25
Use of Filters	33-25
Results of Not Serving Swapping Requests	33-25
Deinstallation of a System Service	33-26
System Service Operations	33-28
Basic Requests Used by all System Services	33-28
System Requests	33-28

Section 34. System-Common Services Management

What is a System-Common Service?	34-1
System-Common Service Model Overview	34-2
Compared to Request-Based System Services	34-3
Similarities	34-4
Differences	34-5
Choosing a System Service to Write	34-7
System-Common Service Writing Guidelines	34-8
Serving Requests	34-8
Serving a Deinstallation Request	34-9
How to Write a System-Common Service	34-9
Calling the System-Common Service	34-11
To Eliminate Linking	34-11
System-Common Service Operations	34-12

Section 35. Extended System Services

What are Extended System Services?	35-1
Extended System Services Operations	35-2
Asynchronous System Service	35-2
CD-ROM Service	35-4
Command Access Service	35-6
Mouse Services	35-6

Performance Statistics Service	35-9
Queue Manager	35-10
Sequential Access Service	35-12
Spooler Management	35-14
Voice/Data Services	35-14

Section 36. Partitions and Partition Management

What Is Partition Management?	36-1
Partitions	36-2
Types	36-2
Fixed or Variable	36-2
Partition Components	36-3
In-Memory Relationships	36-3
Local Descriptor Table	36-5
User Structure	36-5
Program Code	36-7
Short-Lived and Long-Lived Memory	36-7
User Number	36-8
Partition Organization In System Memory	36-9
At System Initialization	36-9
Single Application Partition In Memory	36-11
Multiple Application Partitions In Memory	36-12
Partition Swapping	36-13
Creating a Partition	36-15
Loading a Program	36-16
Terminating a Program	36-16
Removing a Partition	36-17
Obtaining Partition Status	36-17
Communicating Between Application Partitions	36-17
Application Partition With Multiple Run Files	36-18
Partition Management Operations	36-20
Basic Partition Management Operations	36-20
Swapping	36-21
Partition Creating Under Program Control	36-22
Partition Loading Under Program Control	36-22
Loading Additional Tasks	36-23

Section 37. Timer Management

System Timers	37-1
Realtime Clock	37-1
Programmable Interval Timer	37-1

Using the Timer Management Operations	37-2
Using Delay and Doze	37-2
Using ShortDelay	37-3
Realtime Clock	37-3
Timing a Single Interval	37-4
Repetitive Timing	37-5
Programmable Interval Timer	37-6
Timer Management Operations	37-8
Delay	37-8
Realtime Clock	37-8
Programmable Interval Timer	37-8

Section 38. Virtual Code Management

What is Virtual Code Management?	38-1
Overlays	38-2
Writing or Retrofitting Overlay Programs	38-2
Model Overview	38-3
Data Structures	38-4
Overlay Zone Header	38-6
StaticsDesc	38-6
Return Overlay Descriptors	38-7
ProcInfoNonres	38-7
Protected Mode Operation	38-8
Real Mode Operation	38-9
Intercepting Calls	38-9
Intercepting Returns	38-10
Importance of Call/Return Conventions	38-13
Calls to Procedural Addresses	38-13
Adjusting Addresses	38-14
Virtual Code Management Operations	38-16
Basic	38-16
Advanced	38-16

Section 39. Dynamic Link Libraries

What are Dynamic Link Libraries (DLLs)?	39-1
Dynamic Link Library Terminology	39-2
DLLs Compared to Other CTOS Services	39-3
Statically Linked Object Modules	39-4
DLL Procedures	39-4
Comparison of DLLs to System-Common Services	39-4
Request-Based System Services	39-4

How Dynamic Linking Works	39-6
Impact on Loading	39-8
Mapping Linktime to Runtime Addresses	39-9
Module Definition File	39-10
Using the Module Definition Utility	39-11
Using Resources	39-11
Calling DLL Procedures	39-11
Sharing Data Among DLL Clients	39-11
General Guidelines for Writing DLLs	39-12
Observing Caveats	39-13
Defining the DLL and Client Modules	39-14
Executing Initialization Procedures	39-14
Terminating Clients	39-15
Configuring Your System to use DLLs	39-16
Handling Errors	39-16
Setting Up DLL Search Paths	39-17
Defining DLL Segments	39-17
Assigning LDT Selectors	39-17
Summary of DLL Segments	39-19
Special Use of Instance Segments	39-20
Runtime Dynamic Linking	39-20
Requests to Perform Runtime Linking	39-20
DLLs: In Conclusion	39-21
Dynamic Link Library Operations	39-22

Section 40. Interrupt Handlers

Interrupt Handling Terminology	40-1
External Interrupt Handling Model	40-5
Device Handling	40-5
Device Handler Process	40-7
Device Interrupt Handler	40-7
Controlling When External Interrupts Occur	40-8
The Interrupt Flag	40-8
The Programmable Interrupt Controller	40-9
Pending and Lost Interrupts	40-11
Nonmaskable Interrupts	40-12
Operating System Interrupt Handler Styles	40-12
CRIHs and CMIHs	40-15
Guidelines for Writing a CRIH	40-16
Guidelines for Writing a CMIH	40-18

RIHs and MIHs	40-19
Guidelines for Writing an RIH	40-20
Guidelines for Writing an MIH	40-22
Examples of External Interrupt Handlers	40-22
Parallel Port Interrupt Handlers	40-22
X-BUS Interrupt Handlers	40-23
Pseudointerrupts	40-24
Internal Interrupts	40-25
Software Interrupts	40-25
Program Exceptions	40-25
Faults	40-26
Trap Handlers	40-27
Packaging of Interrupt Handlers	40-28
Application Program	40-29
System Service	40-29
Interrupt Handler Operations	40-30

Section 41. X-Bus Management

What is an X-BUS?	41-1
Locating the Base Address	41-1
Dynamically Obtaining the Base I/O Address	41-2
Computing the Module Base Address	41-2
X-BUS Module/Processor Memory Access	41-3
Accessing X-BUS Module Memory	41-3
Using X-Bus Operations to Access Memory	41-3
Specifying a Window Size	41-4
Accessing Modules In Protected Mode	41-5
Accessing Modules In Real Mode	41-5
X-BUS DMA	41-6
Communication and Start-Up Protocols	41-6
XBIS	41-7
X-BUS Interrupts	41-8
X-Bus Management Operations	41-9

Section 42. Bus Address Management

What is Bus Address Management?	42-1
Why Bus Addresses?	42-1
Bus Addresses on an SRP	42-3
Who Uses Bus Addresses?	42-5

Using DMA Buffers	42-6
Piecemealing the DMA Buffer	42-7
Deallocating the DMA Buffer	42-7
Bus Address Management Operations	42-8
 Section 43. Configuration Management	
System Administrative Actions	43-1
Programmer Actions	43-1
 Section 44. Cluster Management	
What is Cluster Management?	44-1
Cluster Environment	44-1
Status	44-1
Polling	44-2
Roll Call	44-2
Repoll	44-2
Request Routing Across the Cluster	44-3
Cluster Management Operations	44-4
 Section 45. Native Language Support	
What is Native Language Support?	45-1
Extended Native Language Support	45-1
Message Files	45-2
NLS Terminology	45-2
How to Take Advantage of NLS	45-2
NLS Tables	45-3
Detailed NLS Table Descriptions	45-7
Keyboard Mapping	45-7
File System Case	45-8
Lowercase to Uppercase	45-8
Video Byte Streams Text	45-8
Uppercase To Lowercase	45-8
Keycap Legends	45-9
Date And Time Formats	45-9
Number and Currency Formats	45-10
Date Name Translations	45-10
Collating Sequence	45-10
Character Class	45-12
Yes or No Strings	45-13
Special Characters	45-13

Contents

Keyboard Chords	45-13
Multibyte Escape Keys	45-13
NLS Strings	45-14
ClusterShare Keyboard	45-14
ClusterShare Keyboard Extended Codes	45-15
ClusterShare Video Translation	45-15
ClusterShare Keyboard Key Post Values	45-15
OFIS Spreadsheet	45-15
Context Manager	45-16
Gengo Date Formats	45-16
Using the NLS Tables	45-17
NLS Tables Loaded at System Initialization	45-17
Alternate NLS Table Sets	45-17
Internationalization	45-18
Extending Internationalization	45-19
ENLS Character Processing	45-20
ENLS String Processing	45-21
ENLS Line/Form Drawing	45-21
Updating Applications	45-22
Localization	45-23
Customizing the NLS Tables	45-23
Linking With the Appropriate ENLS Library	45-23
Other System Customization Requirements	45-24
Message File Facility	45-24
Creating and Editing Message Files	45-24
Message File Operation Sets	45-25
Using Macros With Messages	45-26
Native Language Support Operations	45-28
NLS Utility	45-28
ENLS Utility	45-30
Message File	45-33
 Appendix A. Operating System Features	 A-1
 Glossary	 1

Figures

2-1.	Variable Partiton and Multipartition Systems	2-15
2-2.	Virtual Memory Operating Systems	2-17
2-3.	Application Partition and Free Memory	2-18
2-4.	Memory Organization Under Partition Management	2-20
2-5.	Mapping Pages to Physical Memory Frames	2-21
2-6.	Memory Organization of an Application Partition	2-23
3-1.	Linear Address Space	3-7
3-2.	Oversubscribing Physical Memory	3-8
3-3.	Clock Algorithm	3-10
4-1.	Interface Levels	4-9
4-2.	Memory Address Translations	4-11
5-1.	From Source Modules to Program in Memory	5-2
6-1.	Executive Variable Length Parameter Block	6-5
6-2.	Filled-in Variable Length Parameter Block	6-7
7-1.	Interface Levels	7-2
11-1.	Keyboard Management Overview	11-8
11-2.	General Data Block Layout	11-19
11-3.	Translation Data Block Format	11-21
11-4.	Comparing Data Blocks	11-27
11-5.	NlsKbd.sys Format	11-35
11-6.	System Input Process	11-47
11-7.	Escape Sequence for Entering User Data	11-51
12-1.	Effects of Volume Encryption	12-19
12-2.	Volume Control Structures	12-31

Contents

17-1.	Ports/Access Methods Relationship	17-2
18-1.	Sample SCSI Configuration	18-6
18-2.	SCSI Manager Example	18-24
24-1.	Expand Up and Expand Down Segments	24-5
24-2.	From Source Modules to Program in Memory	24-7
24-3.	Application Partition Memory Organization	24-8
25-1.	Formatting Cache Memory	25-7
26-1.	Adding Resources to a Run File	26-6
26-2.	Resource Descriptor Table	26-9
26-3.	Name Heap	26-19
29-1.	Process States	29-5
30-1.	Client and System Service Interaction	30-2
30-2.	Processing Flow	30-3
30-3.	Communication Between Processes	30-4
30-4.	How IPC Is Used with ICMS	30-5
30-5.	Synchronization	30-6
30-6.	Buffer Management	30-8
30-7.	Request Primitive	30-13
30-8.	Respond Primitive	30-14
30-9.	Send Primitive	30-15
30-10.	Wait Primitive	30-16
30-11.	Sending a Message to an Exchange	30-20
30-12.	Waiting for a Message at an Exchange	30-21
30-13.	Messages Queued at an Exchange	30-22
30-14.	Processes Queued at an Exchange	30-23
30-15.	Request Block for the Write Operation	30-28
30-16.	Requesting to Open a File	30-37
30-17.	Responding With a File Handle	30-38
30-18.	Using the File Handle	30-40
30-19.	Network Routing	30-46
30-20.	Shared Resource Processor Routing	30-48
30-21.	Exchange Routing	30-50
30-22.	Filter Process Interaction	30-51
30-23.	Interprocess Communication Summary	30-52

31-1.	Semaphore Types	31-4
31-2.	Producer/Consumer Model Using Semaphores	31-14
31-3.	Producer/Consumer Model Using Send/Wait	31-15
31-4.	Producer/Consumer Model Using Request/Respond	31-16
31-5.	RAM Semaphore Variable Structure	31-17
32-1.	Z-Block	32-10
32-2.	ICC Interaction	32-13
32-3.	Board A's Wait Exchange Table	32-15
32-4.	Board B's Y-Block Wait Flags Table	32-15
32-5.	Wait Exchange and Wait-Satisfied Flags tables	32-16
33-1.	Client and System Service	33-3
33-2.	Processing Flow	33-5
33-3.	Request Routing Table Fields	33-7
33-4.	Before Conversion to a System Service	33-9
33-5.	Conversion to a System Service	33-12
33-6.	System Service Program Model	33-13
33-7.	One-Way Pass-Through Filter	33-23
33-8.	Two-Way Pass-Through Filter	33-24
34-1.	Request-Based Compared to System-Common	34-3
36-1.	Hypersegments in Protected Mode	36-4
36-2.	Memory Organization at System Initialization	36-10
36-3.	Single Application Partition in Memory	36-11
36-4.	Multiple Application Partitions in Memory	36-13
36-5.	Swapping	36-15
36-6.	Multiple Run Files in an Application Partition	36-19
38-1.	Virtual Code Facility Data Structures	38-5
38-2.	Stub Structure	38-8
38-3.	Tracing the Stack	38-11
39-1.	Execution Models	39-5
39-2.	Imports	39-8
39-3.	Assigning LDT Selectors	39-18

Contents

40-1.	Interrupt Hierarchy	40-4
40-2.	Device Handler	40-6
40-3.	Interrupt Nesting	40-11
40-4.	Interrupt Handler Styles	40-14
40-5.	CRIHs and CMIHs	40-15
40-6.	User-Written CRIH Summary	40-17
40-7.	User-Written CMIH Summary	40-19
40-8.	RIHs and MIHs	40-20
40-9.	User-Written RIH Summary	40-21
40-10.	User-Written MIH Summary	40-22
42-1.	Bus Address Types	42-2
42-2.	Using Bus Addresses on an SRP Board	42-4

Tables

ATM-1.	Common Prefixes	xlvi
ATM-2.	Common Roots	xlvii
2-1.	Workstation Operating System Features	2-8
2-2.	SRP Processor Board Features	2-9
3-1.	Demand Paging Terms	3-2
10-1.	Workstation Video Capabilities	10-12
10-2.	B24 and B27 Workstation Video Capabilities	10-13
10-3.	Character Cell Size	10-14
11-1.	Keyboard Terms	11-2
11-2.	Submit File Escape Sequence Permitted Codes	11-50
11-3.	Action Key Combinations	11-53
12-1.	Protection Levels	12-15
12-2.	Bit Number Designations for Protection Level	12-16
12-3.	Syntax Errors	12-41
18-1.	SCSI Device Management Terms	18-3
18-2.	Information Transfer Phases	18-14
18-3.	Sense Keys	18-19
24-1.	Memory Management Terms	24-1
25-1.	Cacher Terms	25-2
27-1.	System Structures	27-1
29-1.	Process State Transition	29-6

Contents

30-1.	Request Code Levels	30-10
30-2.	Format of a Request Block Header	30-25
30-3.	Interpretation of Resource Handle Bits	30-36
30-4.	Specification Expansion	30-43
30-5.	RtCode Values for Request Routing	30-44
30-6.	RTCode Values for Specification Expansion	30-44
31-1.	Semaphore Terms	31-3
32-1.	Shared Resource Processor Routing Types	32-4
33-1.	RequestTemplate.txt Fields	33-15
33-2.	Creating a Loadable Request File	33-17
34-1.	System Service Comparison	34-7
39-1.	DLL Terms	39-2
39-2.	Service Types Compared	39-3
39-3.	DLL Segments	39-19
40-1.	Interrupt Handling Terms	40-2
45-1.	NLS Terms	45-2
45-2.	NLS Tables	45-4
45-3.	Number and Currency Formats Key Elements	45-11

About This Manual

Audience

This manual is intended for applications and systems programmers who will be writing programs to be run on CTOS operating systems.

Purpose and Scope

This manual guides you through an overview of CTOS operating systems and explains how these systems work. The manual introduces the virtual memory workstation operating system, CTOS III, Version 1.0 for 80386 and 80486 protected mode workstations. In addition, the manual describes the following Unisys operating systems:

- CTOS I, Version 3.4 for real mode workstations
- CTOS II, Version 3.4 for protected mode workstations
- CTOS/XE, Version 3.4 for real and protected mode shared resource processor boards

Collectively, these operating systems are called *CTOS* in this manual. The only time the text calls out a specific operating system name is when a feature unique to that system is being described. In most cases, you can determine whether a feature is supported on your system: if the text does not indicate limited support, the feature is part of the common CTOS software base. You can also consult Appendix A, “Operating System Features.” This appendix cross references main features to the operating systems supporting them.

Two other manuals, the *CTOS Procedural Interface Reference Manual* and the *CTOS Programming Guide*, are very closely related to this manual. The *CTOS Procedural Interface Reference Manual* is the programmer's reference to CTOS. It contains a description of each operation in the System Image and in the standard operating system libraries, *Ctos.lib*, *CtosToolkit.lib*, and *Enls.lib*. The *CTOS Programming Guide* addresses the needs of applications and systems programmers by providing detailed program examples. You can write programs that run on CTOS in several different languages.

The functionality of CTOS can be extended with the dynamic installation of system services. These *extended system services* include the following:

- Asynchronous System Service
- Command Access Service
- CD-ROM Service
- Mouse Services
- Performance Statistics Service
- Queue Manager
- Sequential Access Service
- Spooler
- Voice/Data Services

The section entitled "Extended System Services," summarizes the procedures offered by these services. Details on how to use the procedures in your own programs are contained in the *CTOS Programming Guide*.

Organization of This Manual

This manual introduces you to system concepts and programming interfaces in a logical fashion, starting with those you need to know about or use first. The sections in this manual are organized as follows:

- Section 1 introduces you to the CTOS operating systems: it summarizes those features that are part of the common system software base and describes some of the extended features as well.
- Section 2 provides an overview of the operating system concepts described in detail in later sections. It also provides a review of the memory management styles provided by the operating systems described in this manual.
- Section 3 highlights demand paging, the virtual memory management style used on CTOS III operating systems.
- Section 4 is a collection of practical information and focuses on how to make calls to the operating system. It points out ways to perceive and address memory, indicates levels at which your programs can access devices, and touches upon the issue of standardization.
- Section 5 describes program management, which consists of those operations used by a program to self-load, to self-exit, and to handle error conditions. This subject is presented at a more advanced level in Section 36, "Partitions and Partition Management," which describes how a partition managing program performs comparable operations to manage several programs executing at once.
- Section 6 presents parameter management, a method of passing information from one program to its successor in the same partition.
- Sections 7 through 23 describe how I/O can be performed to devices, such as disks, video, tape, and communication channels.
- Sections 24 through 39 cover operating system theory. These sections describe such subjects as memory and partition management, system services, and how the operating system uses interprocess communication (IPC) and inter-CPU communication (ICC). These sections also present more advanced programming concepts.

- Sections 40 through 42 are related to the I/O sections (Section 7 through 23) but cover the more advanced concepts of interrupts, X-Bus management, and bus address management.
- Sections 43 through 45 are dedicated to the administrative aspects of the operating system. As a programmer, you may not be involved in customizing your system. You may find it beneficial, however, to internationalize your programs so that they can be used on operating systems in other countries.
- Appendix A lists significant operating system features supported by each operating system version. In addition the appendix compares the memory management styles.

The Glossary describes key terms used in this manual and in the *CTOS Procedural Interface Reference Manual*.

What is New in This Manual

The *CTOS Operating System Concepts Manual* describes the following new features introduced with CTOS III, Version 1.0:

1. It describes a new style of memory management called virtual memory management or demand paging. (See Section 3, "Demand Paging.")

Demand paging provides the following operation for application use:

QueryPagingStatistics

2. It introduces new memory management operations allowing the application to allocate global linear address space for dynamic data. Using these operations, an application can manage linear address space addressable by a single selector or by a sequence of contiguous selectors. In the former case, the address space returned can be a huge segment larger than 64K bytes. In the latter, each segment (except the last) is 64K bytes; the last is 64K bytes or less. (See Section 24, "Memory Management.")

The memory management operations are

AllocateSegment
AllocHugeMemory
DeallocateSegment
DeallocHugeMemory
ReallocHugeMemory
ResizeSegment

3. It introduces the name management service. The name management operations allow the application to manipulate strings associated with tags. The service provides multiple unique name spaces. (See Section 26, "Utility Operations.")

The name management operations are

NameRegister
NameQuery
NameTermination
NameRemove
NameGetUniqueClass

Section 26 also introduces resource management. The resource management operations allow an application to dynamically access resources bound into run files such as icons and debugging symbols.

The resource management operations for accessing resources in disk files are

RsrcCopyFromFile
RsrcDelete
RsrcEndSession
RsrcEndSetAccess
RsrcGetAllSetTypeInfo
RsrcGetCountAllSetType
RsrcGetCountSetType
RsrcGetDesc
RsrcGetSetTypeInfo
RsrcInitSession
RsrcInitSetAccess

The resource management operation for accessing resources in memory is

GetModuleResourceInfo

4. It describes semaphores, how they work, and how they fit into the CTOS scheme of message passing. (See Section 31, “Semaphores.”)

The semaphore management operations are

SemOpen

SemClose

SemSet

SemClear

SemWait

SemNotify

SemMuxWait

SemLock

SemClear

5. It describes dynamic link libraries (DLLs) and the process of dynamic linking. (See Section 39, “Dynamic Link Libraries.”)

The DLL operations are

LibFree

LibGetHandle

LibGetInfo

LibGetProcInfo

LibLoad

ExitListQuery

ExitListSet

In addition the manual includes the following changes:

1. It introduces the following operations, which are described in various sections:

AllocCommDmaBuffer

FVSeries

GetBsMember

GetClstrGenerationNumber

GetFileInfoByName

LookUpValue

SetModuleId

2. It compares the three CTOS memory management styles, namely: multipartition (CTOS I), variable partition (CTOS II), and virtual memory (CTOS III). (See Section 2, "Overview of Operating System Concepts.")
3. It provides a separate section on bus address management and DMA buffers. The section describes bus addresses and how to use the DMA buffer mapping operations. The AllocCommDmaBuffer operation (omitted in an earlier manual) is documented here for the first time. (See Section 42, "Bus Address Management.")

Related Documentation

This manual is one of a related manual set that documents the Unisys family of information processing systems. For a description of each manual in the set and for order information, see your sales representative and the *Unisys CTOS Customer Product Information Catalog and Price List*.

The operating system manual set and other documents referenced in this manual are listed below. Programming documentation exists for other separately orderable products. For more information, see the "Related Documentation" sections in the product manuals.

The following manuals are distributed with the operating system. For descriptions, see the *CTOS Operating System Documentation Directory*.

- *CTOS Basic Asynchronous Terminal Emulator User's Guide*
- *CTOS Editor User's Guide*
- *CTOS Executive Reference Manual*
- *CTOS Executive User's Guide*
- *CTOS Status Codes Reference Manual*
- *CTOS System Administration Guide*

The following manual is overpacked with the operating system:

- *CTOS Batch Manager II Installation, Configuration, and Programming Guide*

The following manuals are distributed with the Development Utilities and are also available separately:

- *CTOS Debugger User's Guide*
- *CTOS Programming Utilities Reference Manual: Building Applications* (formerly the Linker/Librarian sections in the *CTOS Development Utilities Programming Reference Manual*). The manual describes, in addition, the Module Definition Utility and the Resource Librarian.)
- *CTOS Programming Utilities Reference Manual: Assembler* (formerly the assembly language sections in the *CTOS Development Utilities Programming Reference Manual*).
- *CTOS Programming Utilities Reference Manual: Customization* (a new manual that includes sections on the keyboard customization utility and nationalization)
- *CTOS/Open Programming Practices and Standards*
- *CTOS Programming Guide*
- *CTOS Procedural Interface Reference Manual*

The following manuals are published by Intel Corporation:

- *iAPX 286 Programmer's Reference Manual*
- *80386 Programmer's Reference Manual*
- *i486 Programmer's Reference Manual*

The following manuals are available with separately orderable products:

- *CTOS Indexed Sequential Access Method (ISAM II) Programming Reference Manual*
- *Graphics Programming Guide*
- *Generic Print System Administration Guide*
- *Generic Print System Programming Guide*
- *CTOS OFIS® Spreadsheet Reference Manual*

The following manuals are published by other outside sources:

- *Small Computer Systems Interface (SCSI-1), ANSI X3.131-1986* (The ANSI authorized standard for SCSI implementation, available through ANSI)
- *Enhanced Small Computer Systems Interface (SCSI-2), ANSI X 3T9.2/86-109* (Revision 5, or above, available through ANSI)

Naming Conventions Used in This Manual

Memory Address

Memory address refers to the logical memory address. (For details on addressing memory, see the section entitled “Using CTOS Operations.”)

Variable Names

The name of the variable implies some of its characteristics. Parameters used in procedure definitions and fields of request blocks and other data structures are named according to this convention.

A variable name is composed of up to three parts: a prefix, a root, and a suffix. Examples of each part are described below.

Prefixes

The prefix identifies the data type of the variable. Common prefixes and the number of bytes required for each are shown in Table ATM-1. Note that a flag passed to the operating system or returned by an operating system procedure is interpreted as meaning TRUE if its value is 0FFh and FALSE if its value is 0. Even if you are using a high-level language that uses a different value for TRUE or FALSE, you must use these values to mean TRUE and FALSE in operating system calls.

Table ATM-1. Common Prefixes

Prefix	Bytes Required	Description
b	1	Byte (character or unsigned integer).
c	2	Count (unsigned integer).
cb	2	Count of bytes (in a string of bytes).
f	1	Flag (TRUE = 0FFh and FALSE = 0).
i	2	Index (unsigned integer).
l	varies	Literal (a constant).
n	2	Number (unsigned integer, same as c above).
o	2	Offset from the segment base address.
p	4	Logical memory address (pointer) consisting of a segment address and an offset.
pb	4	Logical memory address of a string of bytes.
q	4	Quad (unsigned integer).
rg	varies	Array. Usually used with another prefix, for example, the prefix <i>rgb</i> identifies an array of bytes.
mp	varies	Map. A one-to-one correspondence between two variables. Usually used with other variables being mapped, for example, <i>mpcParcRq</i> maps the count of paragraphs (<i>cPar</i>) to the count of requests (<i>cRq</i>) of that size.
s	2	Size in bytes (unsigned integer).
sb	varies	String. An array of bytes where first byte is the size of the string.
sz	varies	String. An array of bytes with a null terminating byte.
w	2	word

Roots

The root can be a unique name for a variable (such as *clientA*), a general name such as the examples shown in Table ATM-2, or a combination of a unique name and a general name (such as *exchClientA*). Common roots and the number of bytes required for each are shown in Table ATM-2.

Table ATM-2. Common Roots

Prefix	Bytes Required	Description
erc	2	Status code
exch	2	Exchange
fh	2	File handle
fhb		File Header Block (FHB)
lfa	4	Logical file address
qeh	4	Queue entry handle
rq	varies	Request block (size varies with the request)
sa	2	Segment address (an sn or an sr)
sg	2	Global Descriptor Table (GDT) selector
sl	2	Local Descriptor Table (LDT) selector
sn	2	Selector (high-order two bytes of a protected mode logical memory address)
sr	2	Paragraph number (high-order two bytes of a real mode logical memory address)
userNum	2	User number

Suffixes

The suffix identifies the use of the variable. Suffixes are

Last	Largest allowable index of an array.
Max	Maximum length of an array or buffer (thus one greater than the largest allowable index).
Ret	Identifies a variable whose value is set by the called process or procedure rather than by the caller.

Examples of Variable Names

cbFileSpec	Count of bytes in a file specification.
ercRet	Status code to be returned to the caller.
pbFileSpec	Memory address of a string of bytes containing a file specification.
pDataRet	Memory address of an area to which data is to be returned to the caller.
ppDataRet	Memory address of a 4-byte memory area to which the memory address of a data item is returned to the caller.
pRq	Memory address of a request block.
sData	Size (in bytes) of a data area.
sDataMax	Maximum size (in bytes) of a data area.
psDataRet	Memory address of a 2-byte memory area to which the size of a data item is returned.
ssDataRet	Size of the memory area to which the size of a data item is returned.

Section 1

Introduction to CTOS

What Is CTOS?

CTOS is a term that collectively refers to the Unisys workstation and shared resource processor operating systems, which are based on the Intel 80X86 family of microprocessors. CTOS is specially designed to provide high connectivity among widely distributed, mission critical applications.

Currently, there are three classes of CTOS workstation operating systems and a separate class for shared resource processors. The workstation operating system classes are

- CTOS I
- CTOS II
- CTOS III

CTOS I is a real mode operating system based on the Intel 80186 microprocessor. It supports real mode operation only.

CTOS II is a protected mode operating system based on the 80286 and 80386 Intel processors. It supports real and protected mode operation. Applications can be run in real or protected mode by switching the processor to the appropriate mode. On 80386-based machines, the Intel virtual 8086 mechanism is used to support PC emulation.

CTOS III is a protected mode operating system based on the 80386 and 80486 Intel processors. Like CTOS II, it supports real and protected mode operation; however, there is no process switching from protected to real mode. To run real mode applications and to support PC emulation, CTOS III uses the virtual 8086 mechanism. The most notable difference between CTOS III and its predecessors is its use of demand paging and virtual memory to run applications.

Finally, CTOS/XE is the class of shared resource operating systems. CTOS/XE consists of real mode and protected mode components. Like the CTOS I class of workstation operating systems, real mode SRP processor boards offer real mode operation only. Like the CTOS II class of workstation operating systems, protected mode SRP boards provide real and protected mode operation with processor switching between modes. The SRP protected mode boards are 386-based.

CTOS Foundation

Key to connectivity is a common software foundation shared by all CTOS operating systems. Protected mode operating systems offer enhancements to this foundation that expand on the numbers of applications that can be supported in mission critical transactions. The virtual memory operating system provides yet another set of enhancements allowing CTOS to explore new regions of connectivity.

The CTOS software foundation consists of features that promote its connectivity capabilities. Alphabetically, these features are

- Event-driven, priority-ordered process scheduling
- Messaged-based operation
- Multiprogramming
- Multitasking
- Nationalization
- Networking built-in

Event-Driven, Priority-Ordered Process Scheduling

Each process (thread of execution) is assigned a priority and is scheduled for execution based on that priority. The Kernel scheduler uses this priority scheme to provide efficient scheduling. In the Executive, for example, the clock process runs at a higher priority than the process accepting user keystrokes.

Scheduling is driven by system *events*. Whenever an event, such as the completion of an I/O operation, makes a higher priority process eligible for execution, that process is scheduled to execute immediately.

This scheduling technique is called *event-driven, priority-ordered scheduling*. It simplifies scheduling and provides faster response times than scheduling techniques that are entirely time-based.

Message-Based Operation

CTOS is message-based. Applications, as well as the operating system, consist of processes, each managing various resources and communicating by means of messages. Overall execution occurs because messages requesting services are dispatched and processed.

Message-based operation permits the dynamic installation and deinstallation of system services without regenerating the system or altering operating system code. Dynamic installation and deinstallation provides the convenience of adding services, such as printing, queue management, the mouse, or windowing support, at any time. Services can be Unisys-provided or user-written.

Multiprogramming

Multiprogramming is the ability to run more than one application in memory at the same time. Multiprogramming supports the independent invocation and scheduling of multiple processes. Additionally, it supports concurrent I/O and multiple processor implementations.

Multitasking (Multithreading)

Multitasking (also called multithreading) is the ability for any application to have more than one process (thread of execution).

The Executive, for example, consists of two processes: one displays the time of day, while a second accepts your keystrokes.

Nationalization

Native language support (NLS) provides a set of utilities, run time libraries, and data structures that can be used for the easy portation of software to run in various languages.

Networking Built-In

Because CTOS is message-based, it automatically has built-in networking capability. Unlike subroutine calls, messages can be filtered and redirected over communications lines for long distances, simplifying the development of distributed and multiprocessing applications.

Protected Mode Enhancements

Enhancement features to the CTOS foundation offered by protected mode operating systems work to expand the number of applications that can be supported in mission critical transactions. Alphabetically, these features are

- Caching
- Extended native language support
- Multiple and international keyboard support
- Protected mode operation
- Real mode application support
- Small Computer Systems Interface (SCSI) management
- Variable partitions with code sharing capability

Caching

Protected mode operating systems support caching of data in main memory. Although file management uses the cacher to store disk file data in memory for convenient access, the operating system allows an application to create its own cache(s) for the storage of any type of data. Provided a shared resource processor includes a protected mode board, the Remote Cache System Service allows caching of disks attached to real mode boards as well.

Extended Native Language Support (ENLS)

The extended native language support (ENLS) interfaces expand CTOS nationalization to allow porting applications that run in native languages with very large character sets.

Multiple and International Keyboard Support

Keyboard management supports multiple keyboards. The default system keyboard file *NlsKbd.sys* supports all the CTOS I-Bus style and the PC-style keyboards (for example, the model, K1, K5, and the SG102-K keyboards). The system keyboard process merely acts on keyboard events based on the keyboard data blocks contained in this file.

Keyboard management supports internationalization. By customizing the contents of the keyboard data blocks, you can reflect different native languages. The default U.S. model K1 keyboard, for example, displays the U.S. dollar sign (\$) when you press the key combination **SHIFT+4**. By making the appropriate changes to keyboard data blocks, you can have this same key combination display the pound sign on the Swedish or English K1 keyboard. Furthermore, through customizations, you can create keyboard data blocks that emulate the key values of an entirely different (target) keyboard that shares few, if any, common key functions with the attached (source) keyboard. (For details on customizing keyboards, see the *CTOS Programming Utilities Reference Manual: Customization*.)

Protected Mode Operation

Protected mode operation provides protection to applications and the ability to run more of them at once. In protected mode, applications can address memory extending beyond the first megabyte up to the maximum allowed by the processor and hardware. On CTOS II operating systems, this frees space in the first megabyte to run more real mode applications. (See “Real Mode Application Support.”) Protected mode system structures can be used to place limitations on the memory applications can access, thereby preventing them from overwriting code or referencing static memory allocated to other applications.

Real Mode Application Support

Real mode applications can run on a protected mode operating system without modifying code, recompiling, or relinking.

To run real mode applications on CTOS II operating systems, the processor mode is changed from protected mode to real mode. In real mode, applications run in the first megabyte of physical memory.

Virtual memory systems use virtual 8086 mode as the mechanism to run real mode applications. Virtual 8086 mode provides each real mode application with one-megabyte of addressability. The paging service maps each application uniquely by means of a local page map to physical memory.

SCSI Management

Through the use of a generic set of SCSI management operations, applications can communicate with any type of SCSI device connected to a workstation or a shared resource processor that has SCSI capability. The SCSI management operations allow your application to use the same logical interface to communicate with all SCSI devices. An application need only concern itself with the characteristics of the device it intends to control. Each SCSI bus can be shared by a number of applications at the same time. The SCSI protocol used to transfer commands and data between devices and applications is totally transparent to the SCSI bus users.

Variable Partitions With Code Sharing Capability

Partitions can change in size dynamically to meet the requirements of the program currently executing. The code of the executing program can be shared by an instance of the same application in a different variable partition, providing efficient memory usage.

CTOS III Enhancements

Additional enhancements to the CTOS foundation offered by CTOS III are described below. Alphabetically, these features are

- Demand paging
- Dynamic link libraries (DLLs)
- Name management
- Semaphores

Demand Paging

On CTOS III operating systems, demand paging allows applications to be given as much memory as they request to run, independently of the amount of physical memory available. The paging service oversees the use of physical memory, ensuring that the 4K byte portions (or *pages*) of code and data currently being accessed are resident in memory. Except for a few programs that must access physical memory, demand paging operates transparently to existing applications and system services, which run without change.

Dynamic Link Libraries (DLLs)

Dynamic link libraries (DLLs) contain procedures that can be shared among several other applications. Unlike conventionally linked libraries whose procedures reside in the caller's run file, DLLs are separate files that are bound to the requesting application or *client* when that client is loaded. Binding modifies the client references to the DLL to point to the DLL as it exists in memory. The DLL contains a list of exports (usually procedures) registered by name and runtime location. When the client refers to the export name, the operating system quickly maps the name to the export address.

DLLs relieve the system of the need to store and load multiple copies of library procedures. Once a DLL is loaded, its code in memory is shared among all clients that call the DLL procedures. Data segments in DLLs may be shared among all the DLL's users or may be unique for each user.

Perhaps the most significant advantage of DLLs over conventional libraries is that, if they require updating to add enhancements or to correct software errors, only the DLL has to be updated and released again. The next execution of the new version of DLL reflects the update. There is no need to change or relink applications using the DLL.

Name Management

Name management provides a set of operations for storing and retrieving data by names associated with tag values. An application can quickly and conveniently look up a tag associated with a name without having to devise its own algorithm to do so. Name management maintains private name spaces to prevent names from colliding with the same names used by other applications.

Semaphores

Semaphores can synchronize processes and ensure exclusive access to shared resources. Because of its distributed, message-based nature, CTOS already has this functionality built in. Semaphores, however, are supported to facilitate porting Presentation Manager applications from other operating systems such as OS/2, which are not message based. Such applications use semaphores extensively.

Section 2

Overview of Operating System Concepts

What is a Process?

A *process* is an independent thread of execution for a program. It carries with it the context (that is, the processor registers, descriptor tables, address translation tables, and so forth) necessary to that thread. One or more processes are created each time a program is scheduled for execution.

The operating system assigns each process a priority to schedule its execution appropriately: priorities range from 0 (highest) to 255 (lowest).

System service processes are processes that manage system resources. All processes, including system service processes, are scheduled for execution in the same way based on their assigned priority.

What Constitutes the Kernel?

The kernel is the most primitive yet most powerful component of the operating system. It provides

- Event-driven priority-ordered scheduling
- Interprocess communication (IPC)
- Inter-CPU communication (ICC)

Event-Driven Priority-Ordered Scheduling

To meet the need for high performance, the operating system kernel provides efficient event-driven priority-ordered scheduling.

Each process is assigned one of 255 priorities and is scheduled for execution based on that priority. Whenever an *event*, such as the completion of an I/O operation, makes a higher priority process eligible for execution, rescheduling occurs immediately. This results in a more responsive system than scheduling techniques that are entirely time-based.

Interprocess Communication (IPC)

The kernel's interprocess communication (IPC) *primitives*, such as Request and Wait (or Check), are the primary building blocks for synchronizing process execution and transmitting information between processes.

Messages and Exchanges

A process can send a message, wait for a message, or poll (check) for a message. When a process waits for a message, its execution is suspended until a message is sent to it, thus allowing processes to synchronize execution. A process can also check to determine if a message is available without suspending its execution.

The operating system is *message-based*. When a process sends a message, it actually sends the message to an exchange rather than directly to another process. Exchanges function as message centers where processes send messages or processes wait or check for messages. Within a single processor, overhead is minimized, because only the address of the message is moved, not the message itself.

A single process can serve several exchanges, in which case it can select one of several kinds of messages to process next. This feature can be used to set priorities for the work the process is to perform.

System Service Processes

The operating system includes a number of system service processes. A system service process receives IPC messages that request the performance of its services. Examples of operating system services include opening or closing disk files, sending output to a printing device, or accepting keyboard input. A process requesting a system service is a *client* process. Any process, including another system service process, can be a client. The use of system service processes and the formalized interface provided by IPC results in a highly modular environment that increases reliability and flexibility.

Note: *Not all system service processes use IPC to serve clients. A different type of system service process that is not request-based is introduced in "System-Common Services Management."*

System services can be linked-in system services in the operating system. The file management system and the keyboard services are examples.

A system service also can be *dynamically installable*. The Queue Manager and electronic mail are examples. Once installed, a dynamically installable system service is indistinguishable in operation from a linked-in service.

Each of the functions provided by the system service can be accessed by a procedural call from a high-level language, such as Pascal or C, as well as from assembly language. The *request procedural interface* masks the complexities of using IPC: it automatically uses a default response exchange and builds the request block message on the stack of the client process.

Kernel primitives also can be called directly. This allows an increased degree of concurrency between multiple I/O operations and computation. The calling process, for example, can perform calculations while it is waiting for other data to be written to a disk file.

System Service Filters

You can customize the function of a system service by writing a filter for that service.

A *filter* intercepts messages destined for another system service. It may modify the effect of the messages, but it does not modify either the calling process or the system service for which the messages were intended.

Inter-CPU Communication (ICC)

Inter-CPU communication (ICC) provides communication between CPUs among the different processor boards on a shared resource processor operating system. (See “Operating System Types” for details on shared resource processors.) The ICC facility is an extension of IPC.

If the requested system service is on the same SRP processor board as the client process, the kernel uses IPC. If, however, the service is on a different processor board, the kernel uses ICC. ICC passes request and response messages between processor boards.

A shared resource processor is compatible with the workstations at the request level. Whether your program runs on a shared resource processor or on a workstation, your program can access system services in the same way (that is, either by using the request procedural interface or by calling the kernel primitives).

Command Interpreter: the Executive

Interaction with the workstation operator is a function of the Executive, not the operating system. This allows you to choose how to use the screen and the keyboard.

The Executive is an interactive command interpreter providing a command driven user interface that includes a HELP facility, command files, and the interactive addition of new commands. The Executive is also a normal application-level program. (For details on the Executive, see the *CTOS Executive Reference Manual*.)

Since the Executive is a separate run file and not part of the operating system, you can optionally replace it with a customized command interpreter of your own design. Most CTOS systems do, however, use the Executive distributed with the operating system.

File Management System

The file management system organizes files by node, volume, directory, and file. A volume (formatted disk) is automatically recognized when it is placed online (mounted). A file can be dynamically expanded or contracted as long as it fits on one disk, and it can be protected by password (optionally encrypted) and protection level number. Concurrent file access is controlled by read (shared), peek (shared), and modify (exclusive) access modes.

While providing convenience and security, the file management system supplies you with the full throughput capability of the disk hardware. This includes reading or writing any 512 byte sector of any open file with one disk access, reading or writing up to 64K bytes (127 sectors) of any open file with one disk access, overlapping I/O with process execution, and optimizing disk arm scheduling.

The duplication of critical volume control structures protects the integrity of disk file data against hardware malfunction. Two volume home blocks are created for each volume. In addition, two file header blocks can be created for each file on a volume. In the Executive, you can use the **Volume Archive** command to recover a file if either of its redundant file header blocks is valid.

If disk space is more important than data protection, the **Format Disk** command can be used to suppress the duplication of volume control structures.

User-Written Device Handlers

The operating system is designed to accommodate user-written device handlers. A device handler can be part of the application program, or it can be a system service. The kernel can either save process context, allowing the use of handlers written in high-level languages, or (CTOS I only) an assembly language interrupt handler can receive the interrupt directly from the hardware. IPC provides an efficient, yet formal, interface from interrupt handler to device handler and from device handler to application program.

Distributed Environment and Clustering

The operating system supports local resource-sharing networks (clusters) as well as standalone workstations. The network extends the operating system resource-sharing capability over a wide area.

Local Resource-Sharing Networks (Clusters)

A *cluster configuration* consists of cluster workstations connected to a server. The *server* can be another workstation or a shared resource processor. Essentially the same operating system executes in each cluster workstation as in the server workstation (or in the combined processors of a shared resource processor). The server provides resources, such as the file system and queue management, for all workstations in the cluster. Concurrently, a server can support its own interactive application program processing.

In the cluster configuration, the IPC facility of the operating system is extended to provide transparent access to system services that execute at the server. While some services (such as queue management, 3270 emulator, and database management) would be installed at the server, others (such as video management and keyboard management) remain at the cluster workstation. A cluster workstation with its own file system can service file requests locally as well as send file requests to the server.

One high-speed, RS-422 or RS-485 cluster communications channel is standard on each workstation. In a cluster configuration connected to a server workstation, the server and all of the workstations connected to it use this channel for communications with each other. For clusters with a shared resource processor server, multiple channels are provided.

Network

The network extends the operating system resource-sharing capability. It allows resources such as the file system, ISAM, X.25 Network Gateway, and printing services to be shared among workstations connected by communications lines over a wide area, as well as among workstations in the local cluster.

Operating System Types

Operating systems are available for workstations and shared resource processors.

Workstation operating systems are of the following types:

- Server workstation
- Cluster workstation without a local file system
- Cluster workstation with a local file system

Shared resource processors are multiprocessor machines consisting of one to several SRP processor boards. Each type of board supports a different System Image. The board types are listed below:

- General Processor (GP)
- General Processor with Communications Interface (GP+CI)
- General Processor with SCSI Interface (GP+SI)
- Cluster Processor (CP)
- Data Processor (DP) [Storage Processor (SP) plus Storage Controller (SC)]
- File Processor (FP)
- Storage Processor (SP)
- Terminal Processor (TP)

Workstation Operating Systems

Table 2-1 summarizes features available on each workstation operating system.

Table 2-1. Workstation Operating System Features

Operating System	Workstation Agent	Mstr Agent	File System
Server		X	X
Cluster without local file system	X		
Cluster with local file system	X		X

The differences between each workstation operating system are a function of the services each has to offer.

The cluster workstation operating system (optionally) excludes the file management service and the disk handler, and includes a workstation agent. The workstation agent transmits local requests over the cluster line to the server. A cluster workstation with a local file system includes a file management service.

The server workstation operating system differs from a cluster workstation in its inclusion of a master (Mstr) agent. The master agent polls the attached cluster workstations for requests that it queues at the specified exchanges at the server. The server workstation can provide file services for the entire cluster configuration.

Shared Resource Processor Systems

A shared resource processor system supports several different System Images each running on a unique processor board. System Images are named according to the processor board upon which they run and whether the processor is a real mode (r) or protected mode (p) processor, for example, pSrpGp, rSrpCp, or rSrpFp. Each processor board contains a CTOS kernel and memory, and generally provides a subset of the services offered by a workstation operating system. Services provided by individual boards can be shared among all others. Interboard communication is achieved by means of a high-speed bus using the ICC facility. Together, the System Images function as a unified operating system.

A minimal shared resource processor configuration consists of either

- One GP+SI processor board
- At least one FP or DP and one CP

Table 2-2 summarizes features provided by each SRP processor board.

Table 2-2. SRP Processor Board Features

SRP Pro- cessor Board	Pro- cessor Type RS-422 Ports	Mstr Agent RS-485/	RS-232-C Ports Tape	File System Ports	Half- Inch	V.35/ X.21
GP	386	X	X			
GP+CI	386	X	X			X
GP+SI	386	X	X	X		
FP	186			X		
DP	186			X	X	
SP	186				X	
CP	186	X	X			
TP	186		X			

The GP+SI, FP, and DP provide file management services, differing only in the type of hardware upon which the service is performed. The GP+SI services SCSI hard disks and QIC tape drives, whereas the FP services the ST 506 class of disk drives. Along with servicing SMD disk drives, the DP supports half-inch tape. The SP handles half-inch tape exclusively.

All of the following processor boards contain peripheral communications ports for cluster or network communications:

- GP
- GP+CI
- GP+SI
- CP
- TP

Each of these processor boards provides an RS-232-C communications service to support asynchronous terminals and network communications media. The GP+CI contains 6 additional RS-232-C ports, which can be used for high speed communications: two can be configured as V.35 ports and two as X.21 ports. Each processor board in this group except the TP provides a Master Agent to transport messages over cluster lines to local workstations.

Programs and Run Files

An executable *program* can consist of code, data, and one or more processes. Programs are loaded into memory from disk-resident files called *run files*. A run file is created by compiling and/or assembling source language modules into object modules and linking the object modules together into code and data segments.

Setting Up the Operating Environment

When a currently active program such as the Executive requests it to do so, the operating system reads a specified run file into memory and prepares the program to run. Setting up the operating environment for a program includes such tasks as assigning segment selectors (protected mode programs), relocating intersegment references, and scheduling the program for execution.

Partitions and User Numbers

In the operating environment, all the operating components and resources associated with the executing program (such as its code, static data, dynamic memory, open files, and so forth) are associated with a unique *user number*. The user number is shared by all the program processes.

User numbers were previously known as partition handles. Because earlier operating systems loaded programs into statically fixed memory areas of a prescribed size, it made sense to refer to the code or open files by the handle of the partition.

With the development of more recent operating systems, however, partitions became less static entities. Not only did partitions vary in size and memory location but also the components of the partition (code, data, and so forth) could be loaded independently anywhere in memory. (See “Memory Management Styles” for details.) On these systems it made more sense to refer to the user of the partition in calls to the operating system by a unique, identifiable user number.

In a cluster or network environment, the resources of each cluster workstation partition are identified at the server by a user number that has been translated to be unique for that server.

(See “Swapping,” for additional information on user numbers and swapping programs.)

Memory Management Styles

Three styles of memory management are represented historically by the operating systems described in this manual. From earliest to latest, these styles and the operating system classes to which they pertain are

- Multipartition (CTOS I)
- Variable partition (CTOS II)
- Virtual memory (CTOS III)

With the introduction of each new style, the partition has assumed a less important role.

Although this manual frequently depicts the components referred to by a user number as all residing in a memory partition, this is done as a simplification. (See “Partitions and User Numbers.”) The memory management style determines whether the partition components really exist contiguously in memory. (For a comparative summary of memory management styles and the operating system versions to which they apply, see Appendix A, “Operating System Features.”)

Multipartition Memory Management

On multipartition real mode operating systems (CTOS I), the partition is almost an atomic unit. Partition size and location are fixed in physical memory.

Code, data, partition memory, and supporting structures for an application are all contiguous and deliberately placed within the partition in physical memory. To make room to run more applications, the entire partition is swapped to disk. (See “Managing Physical Memory,” for details on swapping.)

Variable Partition Memory Management

On variable partition protected mode operating systems (CTOS II), the partition is composed of individually swappable units called *hypersegments*. A hypersegment is a contiguous region of memory of arbitrary size containing partition information, for example, the code or the local descriptor table for an application. The advantage of hypersegments over partitions is that they are smaller, making them easier to manipulate.

The operating system loads hypersegments into physical memory wherever they fit, and swaps them to disk, as necessary, to create room for other hypersegments.

To add flexibility in managing memory, variable partition systems support fixed and variable partitions. A *fixed* partition always uses a fixed amount of memory. A *variable* partition can use up to the maximum amount of memory that the program executing in it may allocate. (For details, see the Linker documentation in the *CTOS Programming Utilities Reference: Building Applications*.) Depending on the amount of available memory, the size of a variable partition can grow up to the maximum size specified. Variable partitions permit program code to be shared by an instance of the same program in another variable partition.

The term *user number* is first introduced with variable partition operating systems for convenience in referencing related hypersegments. (See “Partitions and User Numbers.”)

Virtual Memory Management

Virtual memory protected mode operating systems (CTOS III) mark the obsolescence of partitioned memory. On virtual memory systems, the paging service oversees use of physical memory. Only those 4K byte portions of a hypersegment that are in use reside in physical memory.

Furthermore, the memory management interfaces introduced with this operating system (for example, `AllocHugeMemory` and `AllocateSegment`) allow the application to obtain address space dynamically from the global linear address space rather than from partition memory. (See “Virtual Memory Management,” for details on global linear addresses.)

Memory Organization at System Initialization

To see how system memory is organized, we first look at the linear address space. The linear address space is also called the virtual address space in other documentation.

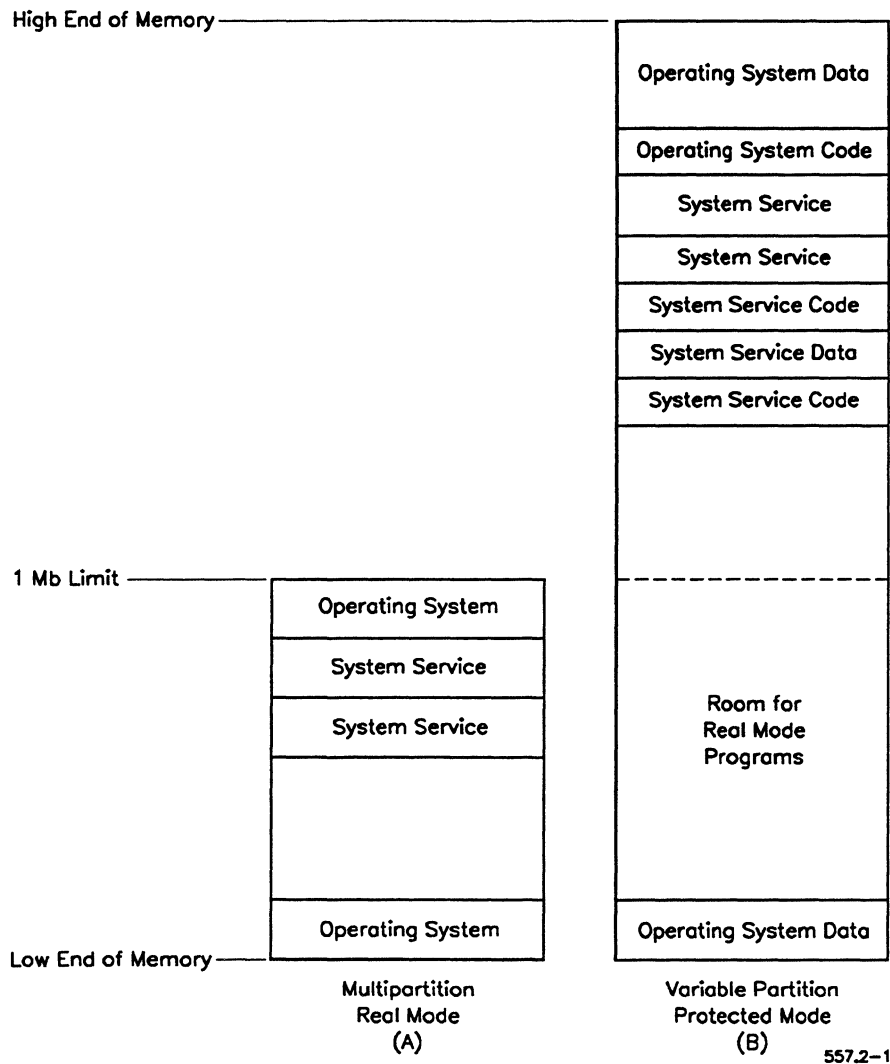
On multipartition and variable partition operating systems, the linear address space is the same as the physical address space. (For a discussion of memory translations, see “Addressing Memory” in the section entitled “Using CTOS Operations.”) On these systems, the amount of addressable memory is determined by the processor and hardware limitations.

On virtual memory systems, the linear address space is 4 gigabytes. Typically it is much larger than the physical address space. On these systems, linear addresses are not equivalent to physical addresses.

Variable Partition and Multipartition Operating Systems

Figure 2-1 shows the linear address space at initialization for multipartition real mode and variable partition protected mode operating systems.

Figure 2-1. Variable Partiton and Multipartition Systems



At (A) in the figure, we see the multipartition operating system layout. The maximum linear address space is one megabyte. This is all the memory a real mode program can address.

The figure shows a few system partitions. After the operating system and system services are loaded, the remaining memory is available to load applications.

At (B) in the figure, we see the system memory layout for a variable partition protected mode operating system. Variable partition systems differ from multipartition systems in the following ways:

- The linear address space usually is larger. (Typically the hardware provides more physical memory to take advantage of the greater address range provided by the processor.)
- The hypersegments of a partition or user number can be loaded independently, anywhere in memory.
- To run as many real mode programs as possible, most operating system segments and all system services are placed at addresses above the first megabyte.

Virtual Memory Operating Systems

The linear address space on virtual memory management systems differs from either of the previous systems discussed (see “Variable Partition and Multipartition Operating Systems”) in that it is determined by the processor only. Currently, the linear address space is 4 gigabytes. There is much less physical memory.

Programs in the virtual memory environment are allocated linear addresses independently of the amount of physical memory available. The paging service oversees the use of physical memory so that only those portions (*pages*) of code and data currently in use reside in physical memory.

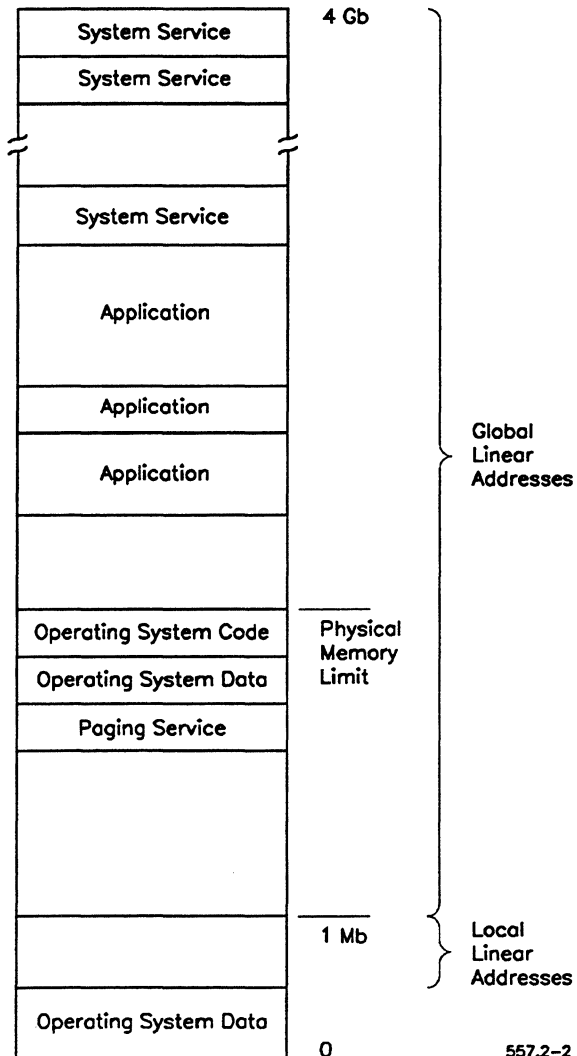
Figure 2-2 shows how the virtual memory operating system organizes the linear address space. The operating system and paging service are at linear addresses in the physical addresses range. The remaining linear addresses in the physical memory range are unoccupied.

By default, the operating system allocates all the linear address space a system service or application needs. The figure shows applications and system services at linear addresses beyond the physical address range.

Global linear addresses are allocated to all protected mode applications and system services starting from the high end of the linear address space and proceeding towards the low end.

Local linear addresses are available for real mode applications to share. The addresses have a unique mapping to physical memory for each real mode application.

Figure 2-2. Virtual Memory Operating Systems



557.2-2

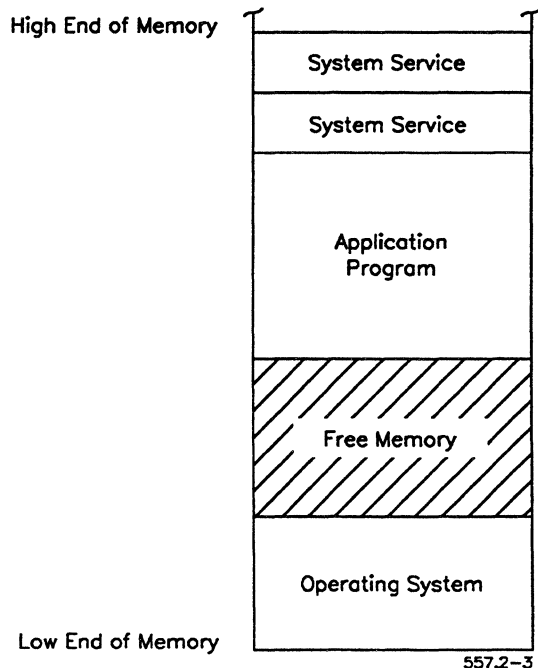
Loading Applications

After initialization, multipartition systems always create partitions in fixed memory positions and load applications into them. Variable partition and virtual memory systems, however, load partition hypersegments wherever they fit in memory.

Bringing Applications into Memory

To bring an application into memory, the operating system creates an application partition in available memory and loads the program into it. (See Figure 2-3.) If the program loaded is a partition managing program, it can create additional application partitions in the free memory and load programs into them. Context Manager is such an example. (For details on Context Manager, see your Context Manager manual.)

Figure 2-3. Application Partition and Free Memory



Managing Physical Memory

To run more programs than there is physical memory space, variable partition and multipartition operating systems use swapping. Virtual memory systems uses demand paging. Each of these techniques is discussed in the paragraphs below. (For a comparative summary of these techniques, see Appendix A, “Operating System Features.”)

Swapping

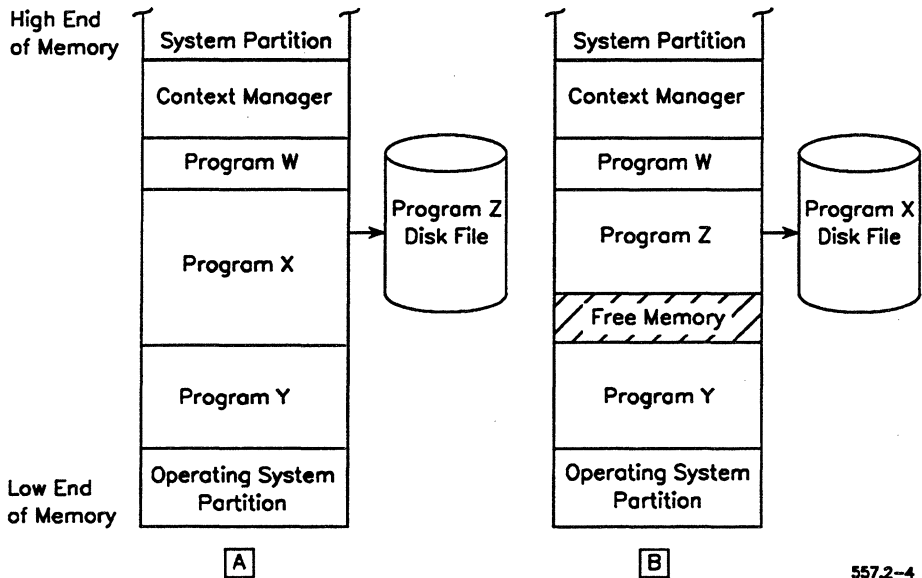
When space is needed on variable partition and multipartition systems, partitions (or user numbers) are swapped out of physical memory to a disk file.

On multipartition systems, the entire partition is swapped. Variable partition operating systems swap hypersegments of a user number, as required.

Figure 2-4 shows swapping on a variable partition operating system. At (A) in the figure, we see Context Manager with Program W, Program X, and Program Y in physical memory and Program Z swapped to disk. At (B), Program X is swapped out, and Program Z is swapped in.

Each partition shown in the figure is associated with a different user number. (See “Programs and Run Files.”) Note that while Program Z’s partition is in the same physical location that Program X’s partition occupied when it was resident in memory, its user number is different. Program X’s user number, however, can be used to refer to Program X, even when the program is swapped to a disk file.

Figure 2-4. Memory Organization Under Partition Management



557.2-4

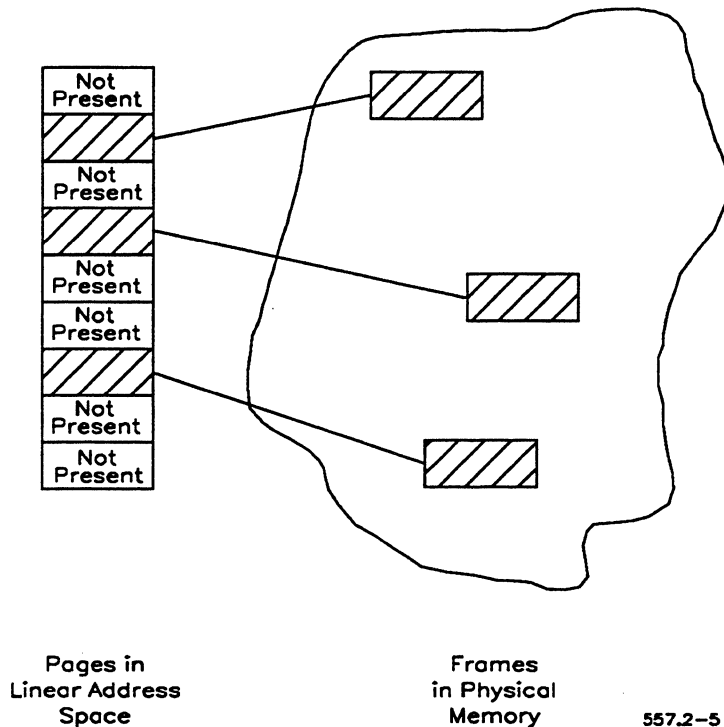
Demand Paging

Virtual memory operating systems use demand paging to provide an operating environment in which programs are given as much address space as they request to run, independently of the amount of physical memory available.

The paging service oversees the use of physical memory. Only those portions of code and data currently in use reside in memory. The paging service maps 4K byte pages in the linear address space to page *frames* in physical memory.

See Figure 2-5. Pages can be mapped to locations anywhere in the physical address space, or they may never be in physical memory if they are not used.

Figure 2-5. Mapping Pages to Physical Memory Frames



If the total number of pages allocated to applications and services is greater than the number of available frames, physical memory is said to be *oversubscribed*. Paging intentionally supports this condition. If, as a result of oversubscription all page frames are in use, the paging service replaces the contents of frames with pages currently needed. Oversubscription works as long as all programs can make reasonable forward progress.

Except for the few cases where applications need to know about physical memory addresses, demand paging operates transparently to existing applications and system services, which run without change.

Virtual Code Management

The virtual code management facility permits the execution of applications exceeding the physical memory of the application partition on multipartition and variable partition operating systems. Virtual code management does this by maintaining portions (object modules) of an executable program in relocatable overlays, which it reads into memory as they are needed.

Unlike demand paging, overlays are not transparent to the user. The programmer sets up the controls to ensure optimal performance.

Virtual code management was the first implementation of “virtual memory” and is supported by all versions of CTOS.

Note: *When an overlay program is run on a virtual memory operating system, the system treats the overlays like ordinary program parts. Whether or not code is placed in an overlay, it resides in memory while it is in use.*

Application Partition Memory Organization

The two types of memory allocation available to an application program are short-lived and long-lived. Within each application partition, *short-lived* memory expands downward from high memory locations, while *long-lived* memory expands upward from low memory locations. (See Figure 2-6.)

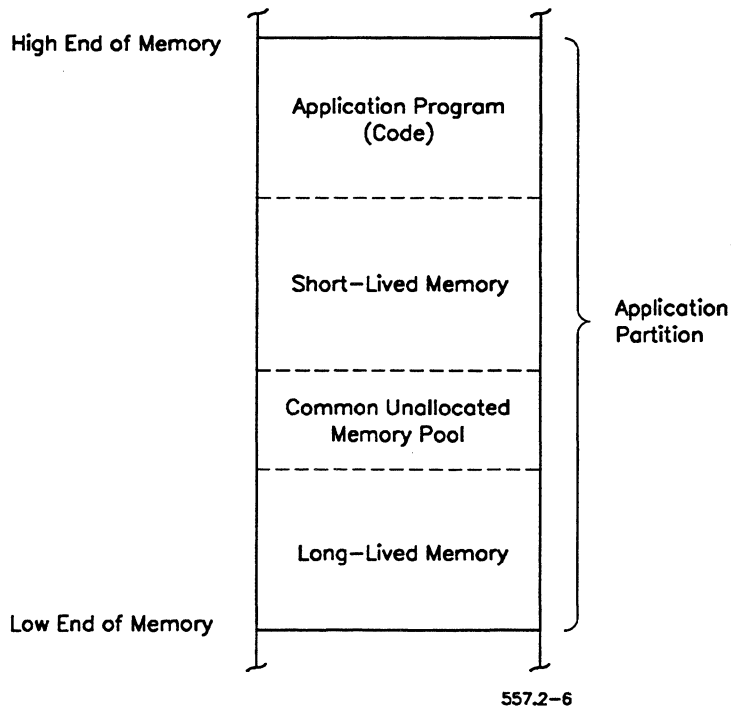
Applications can allocate and deallocate short-lived and long-lived memory by making operating system requests. An application in one partition cannot allocate or deallocate memory in another partition.

Relative to allocations from one end of the memory of an application partition, deallocations must occur in exactly the opposite sequence. That is, the user must follow a last-allocated, first-deallocated discipline when deallocating either long-lived or short-lived memory. For example, if a program allocates short-lived memory segments A, B, and C, it must deallocate them in the order C, B, A.

An application allocates short-lived memory to hold information it needs while executing. For example, it may build a record structure in short-lived memory. Short-lived memory cannot be used to pass information to other partitions. When the execution of an application is terminated, the short-lived memory of its partition is automatically deallocated.

Long-lived memory is deallocated only at the specific request of the program. It is, therefore, useful for passing information from one program to another in a partition, even when the two programs do not coexist. The Executive uses long-lived memory for passing parameters to application programs that will run in the same partition. The Executive typically deallocates long-lived memory whenever it is reloaded.

Figure 2-6. Memory Organization of an Application Partition



Dynamic Linking

Virtual memory operating systems support the dynamic linking of libraries to clients. Unlike conventional linking which copies procedures directly into the caller's run file, dynamic linking loads the dynamic link library (DLL) as a separate file. Only the addresses of called procedures (not the actual procedures) are written to the procedure calls in the client program. Dynamic linking relieves system memory of multiple copies of the same library procedure.

Code and Data Sharing

Code is shared by programs that call DLL procedures. Once a DLL is loaded into memory, the code it contains is shared among the clients that call it.

DLL data may be shared among all programs calling the DLL, or each caller can have its own copy of data. For details, see "Impact on Loading" in the section entitled "Dynamic Link Libraries."

Section 3

Demand Paging

Demand Paging Overview

Demand paging provides a virtual memory environment in which programs are allocated address space independently of the amount of physical memory available.

The paging service maps 4K byte portions of a program called *pages* in the linear address space to page *frames* in physical memory. If necessary, the service copies the contents of the page from the disk to the frame.

With the exception of applications that must access physical memory such as those that directly program the DMA hardware, demand paging operates transparently to existing applications and system services, which can run without change. (Special DMA mapping operations ensure contiguous physical memory for DMA buffers. For details, see the section entitled “Bus Address Management.”)

Demand Paging Terminology

Terms relating to demand paging are described in Table 3-1. You may need to refer to this table from time to time as you read about paging.

Table 3-1. Demand Paging Terms

Term	Definition
Backing store	A run file or swap file where modified pages reside when they are not in RAM.
Clean page	A page in memory that is not modified.
Dirty page	A page that has been modified in memory and must be cleaned if it is replaced.
Frame	A 4K byte region of physical memory.
Global thrashing	Frequent backing store activity due to page faults that are generated when the sum of the working sets is greater than the number of available frames.
Global linear address	A linear address valid for all users.
Local linear address	A linear address valid for a single user.
Local thrashing	Frequent backing store activity generated by a single application when the application working set is greater than the partition size or the number of available frames.
Locked page	A page in physical memory that is not subject to replacement.
Oversubscribed	Physical memory condition in which the size of all programs in the system exceeds the size of physical memory.

continued

Table 3-1. Demand Paging Terms (cont.)

Term	Definition
Page	A 4K byte section of a program in the linear address space.
Page cleaning	Writing the contents of a modified page to backing store.
Prefaulting pages	Loading pages not specifically accessed into frames. Prefaulted pages are contiguous with pages that are being accessed and must be faulted in. The paging service prefaults pages anticipating that they, too, will be accessed.
Working set	Number of pages a program needs to make reasonable forward progress.

Demand Paging: A Memory Management Style

Demand paging is a memory management style. It is perhaps the most significant feature difference between CTOS III and earlier operating system versions. Understanding demand paging in the light of the existing styles will help you see how it fits into the CTOS story described in this manual. (For an overview of the CTOS memory management styles, see “Memory Management Styles” in the section entitled “Overview of Operating System Concepts.” Also see Appendix A, “Operating System Features.”)

Like the multipartition and variable partition memory management, demand-paged, *virtual memory management* allows more applications to execute than there is available space in physical memory. *Virtual memory* refers to the fact that applications can use more memory than is physically available. Pages are made available *on demand*. As applications execute, the RAM they need is faulted in.

When memory is oversubscribed, multipartition and variable partition systems swap one or more programs to backing store. The granularity of multipartition swapping is the entire partition. Variable partition memory management swaps hypersegments, for example, all of data or all of code.

Demand paging replaces partition swapping entirely. The paging mechanism allocates address space and replaces pages no longer needed in memory to make room for currently needed pages. Optimizations include routinely writing modified pages to disk (cleaning) and page prefaulting. The finer granularity of the paging unit (4K bytes) combined with prefaulting results in a more flexible use of physical memory than either multipartition or variable partition memory management provides.

Benefits of Demand Paging

Demand paging provides the following benefits over the multipartition and variable partition memory management styles:

- It prevents sandbars. Sandbars are created in physical memory when a hypersegment fragmenting free memory cannot be swapped out. On multipartition or variable partition systems, non-relocatable applications such as PC Emulator can become sandbars.
- It allows more programs to run. On multipartition and variable partition systems, system services reside in memory in their entirety and cannot be swapped out. The demand paging aspect of virtual memory management requires only those portions of a program that are currently executing to be memory resident, thereby freeing memory for more system services or additional applications to run.
- It allows programs to run that are larger than all of physical memory on a system.
- It improves performance on a context switch of applications running under Context Manager because hypersegment or partition swapping is not required.
- It may allow systems to be configured with less memory. Multipartition and variable partition memory configurations require a sum total of memory to run the operating environment, all the required services, and the applications.
- It is transparent to non-DMA programming applications and system services.

Page Mapping

The paging service maintains the status of all page frames in the system in paging structures. These structures are features of the Intel microprocessor. (For details, see the Intel documentation listed in “About This Manual.”) Except for the description of the page table entry (below), this section generally refers to the paging structures as a *page map*.

The page table entry describes the status of a page. If, for example, the page is in physical memory, the entry maps the linear address of the page to the frame address in physical memory. In addition, it contains flags indicating whether or not the page

- Is present in physical memory
- Has been accessed
- Is dirty (modified while in memory)

The operating system and most protected mode programs use a single *global page map*, which contains a page table entry for every page in the system. Real mode applications and PC Emulator use a local page map.

A *local page map* is identical to the global map except the first megabyte of linear address space has a unique mapping to physical page frames. (See “Executing Real Mode Applications.”)

Hardware uses the page map to automatically trace the physical location of code and data as each instruction executes. When a page is not present, the hardware generates a page fault, which is fielded by the paging service.

Allocating Linear Address Space

Figure 3-1 shows how the operating system organizes the linear address space. The figure calls out the following:

- Upper limit of physical memory
- Global linear addresses
- Local linear addresses

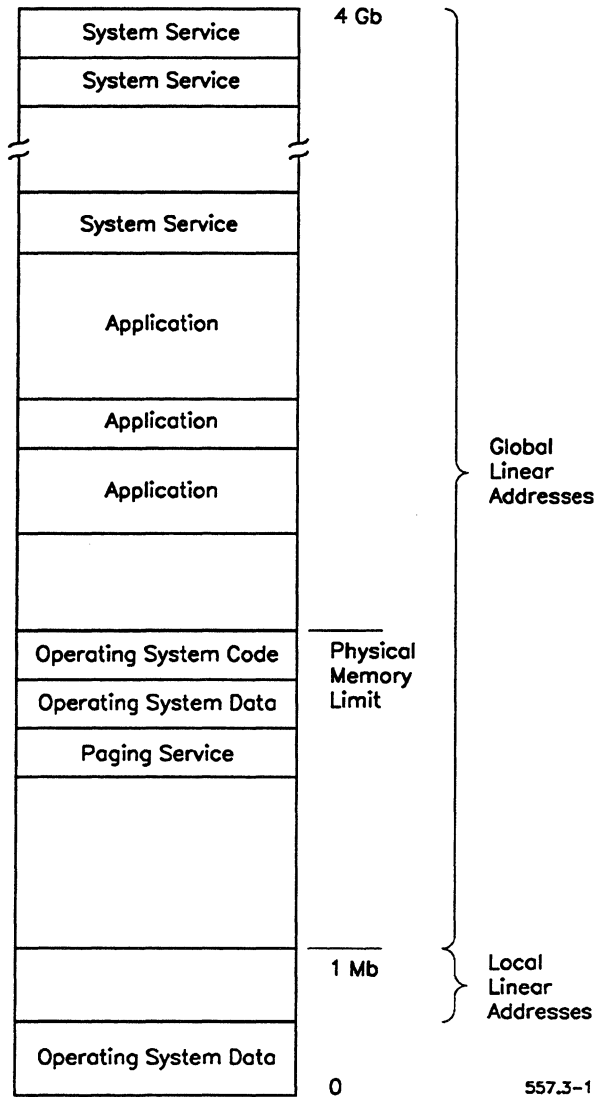
Because the operating system and paging service permanently reside in physical memory, they are at linear addresses within the physical memory range. The remaining linear addresses in this range are unoccupied.

By default, the operating system allocates all the linear address space a system service or application needs. The figure shows applications and system services at linear addresses beyond the physical address range.

The *local linear addresses* are the linear addresses available in the range of 0 to 1 megabyte after the operating system is loaded. The local linear address space is unique to each real mode application.

The *global linear address* space is shared by all protected mode applications and system services.

Figure 3-1. Linear Address Space

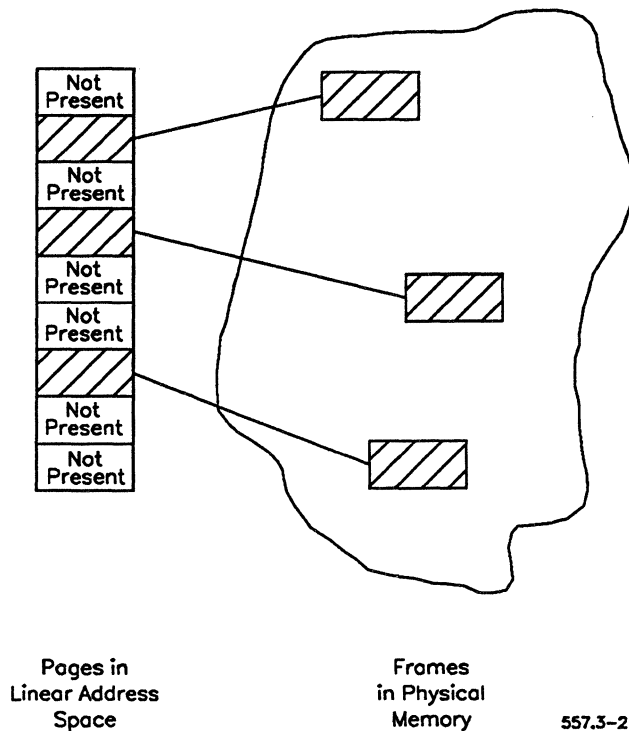


Oversubscribing the Physical Address Space

When the total number of pages allocated to applications and services is greater than the number of available frames, physical memory is said to be *oversubscribed*. This is shown in Figure 3-2.

The pool of available frames is shared by all programs. To bring pages currently being requested into physical memory, the paging service processes page faults. If all frames are in use, the paging service must replace the contents of selected ones. To select pages to replace, it uses a procedure that approaches an LRU algorithm. (For details, see “Replacing Pages.”)

Figure 3-2. Oversubscribing Physical Memory



Paging is intended to support memory oversubscription. Problems can arise, however, if there isn't enough physical memory for all the programs to run effectively.

Basic to every program is the working set. The *working set* is the number of frames a program needs to make reasonable forward progress. If the sum of all working sets is greater than the number of available frames, *global thrashing* can occur, as programs compete against each other for use of the frames. The greater the discrepancy between working sets and frames, the more time that is spent in accessing the disk. As a result, performance can be severely impaired. If physical space is at a premium, the number of programs being run should be reduced to the point where thrashing does not occur.

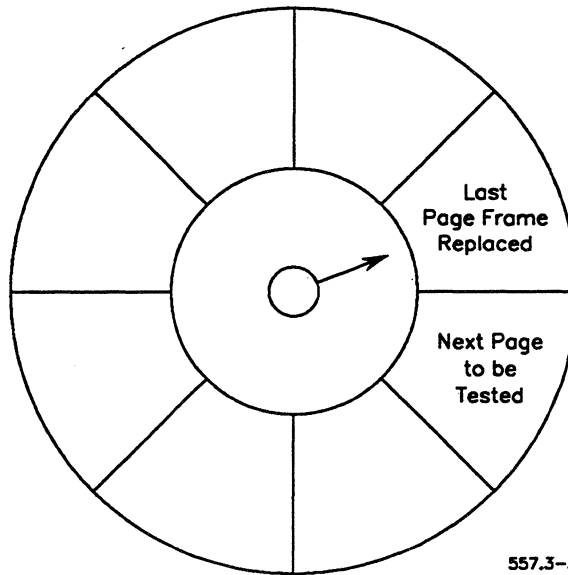
If a program is known to consume excessive amounts of physical memory, you can control system performance by allocating frames based on partition size. (For details, see "Tuning Paging Through System Configuration.") If the working set of an application is greater than the partition size, this condition will result in *local thrashing*, as the application competes against itself for partition frames.

Replacing Pages

The operating system uses a simplified version of an LRU algorithm to replace pages. To lessen the overhead of a true LRU, it doesn't maintain linked lists or time stamps. As such, it may not select the least recently used page to replace. Nevertheless, it achieves comparable performance results.

The replacement algorithm is called the clock algorithm. The frames for each application are arranged in a circular list like the numbers around a clock. The clock pointer (or hand) points at the last page replaced and advances clockwise to the next frame when the algorithm is invoked. (See Figure 3-3.)

Figure 3-3. Clock Algorithm



When invoked, the algorithm advances to the next frame and tests it for replacement. (The algorithm tests and resets the accessed flag in the corresponding page table entry. See “Page Mapping.”) If the frame has been accessed since the last time it was tested, it is considered part of the current working set and is not replaced. In this case, the pointer advances to the next frame, which is tested in the same fashion. The procedure is repeated with each successive frame until a frame that has not been accessed is found. That page is eligible for replacement.

If, however, a frame containing a replaceable page is not found, the algorithm must examine frame allocation and use. If it determines that the faulting application is using less than its maximum frame allocation, it attempts to steal a frame from the clock of another user. It selects the clock with the least activity (avoiding the foreground application clock). If, in testing the clock, it still doesn't find a page to replace, it selects and tests another application's clock. It repeats the procedure of selecting and testing until it either finds a frame or it exhausts all the clocks in the system. If, at this point, it still doesn't find a frame, it must replace the contents of one in the faulting application's clock.

Only when an application reaches its maximum frame allotment does the algorithm start to replace pages in the application's own clock. This causes frames to migrate from idle memory users to those currently needing memory.

If all page frames in the system are in use, the cleaning queue can be checked for replacement pages. In this situation, the paging service must wait for a page to be cleaned before replacing it. (See "Cleaning Pages.")

Cleaning Pages

Before a page can be replaced, it must be cleaned or written to backing store. As a performance optimization, the paging service routinely checks to see if pages in memory frames are dirty. It places any dirty pages it finds in the cleaning queue. In the absence of routine cleaning, every time a page is replaced the two-step process of writing out the dirty page and reading the new page into the frame would have to be performed. If the page to be replaced is already clean, only the read procedure needs to be performed.

Although cleaned pages may be modified again before they are actually replaced, routine cleaning reduces the overall effort expended on servicing page faults.

Prefaulting Pages

By default, the paging service prefaults pages into memory. *Prefaulting* brings pages not accessed into physical memory frames in the likelihood these pages will be needed. It is a performance optimization to decrease the number of page faults.

The prefault algorithm prefaults a minimum of 2 pages and up to a maximum of 14. If the application already has the maximum number of pages allocated to its local clock, only 2 pages are prefaulted. Chances are unlikely performance will improve by removing local clock pages just to replace them with other pages of the same ("might be accessed") stature.

The prefault algorithm checks entries in the page table that precede and follow the page to fault in. To prefault pages, the paging service must be able to allocate contiguous frames.

The paging service maintains statistics on the number of pages prefaulted for each user, the number of prefaulted pages that are used, and the number of prefaulted pages that are not used. This information can be accessed dynamically. (See “Querying Paging Statistics.”)

Page Locking

Certain types of pages are not subject to replacement. Request blocks, by policy, are locked into memory until the response information is returned to the client. Because DMA data must be maintained contiguously, the pages of DMA data are locked into frames in memory.

To make room for a contiguous frame sequence such as might be required for a DMA buffer, the paging service may move frames to other memory locations or write out the contents of the frames to backing store.

Paging Service Components

The paging service consists of the following three components:

- Page fault handler
- Paging process
- System-common procedures

At the hardware interrupt level, the page fault handler is invoked when a page fault occurs. The fault handler saves the state of the program containing the faulting page and resolves the fault if it can do so readily. Otherwise, it passes the fault information on to the paging process in a message.

The paging process is a message-driven CTOS process. It handles messages such as page faults from the interrupt handler. In addition, it processes request and response messages that, for example, are associated with mapping pages to physical memory or with reading pages from a run file.

The paging system-common procedures perform routines such as locking outstanding request blocks into physical memory.

Executing Real Mode Applications

Instead of changing the processor mode from protected to real mode to execute real mode applications, virtual memory systems operating systems use virtual 8086 mode. In virtual 8086 mode, each real mode application executes in a unique local linear address space in the first megabyte.

Each real mode application requires its own local page map. The local page map is identical to the global map except that addresses in the first megabyte of the linear address space are uniquely mapped to physical page frames.

On a virtual memory operating system, all programs (except real mode applications) execute in protected mode and have global linear addresses. When a real mode application passes its data buffers in a request block to a protected mode system service, the Kernel aliases the local linear addresses of the buffers to global linear addresses the system service can access. (For details on how aliasing works, see the section entitled “The Segment Descriptor Tables,” in the *CTOS Programming Guide*.)

Querying Paging Statistics

`QueryPagingStatistics` can be called to obtain paging statistics. Either user paging statistics or system paging statistics may be returned by this request. If a user number is specified to `QueryPagingStatistics`, user paging statistics are returned in the format of a User Paging Statistics Record. If `0FFFFh` is specified instead of a user number, a System Paging Statistics Record is returned.

For details, see the description of `QueryPagingStatistics` in the *CTOS Procedural Interface Reference Manual*.

Tuning Paging Through System Configuration

The paging service has a number of configuration options you can specify to tune system performance. Most of the options are of special interest to advanced users. We mention one option here for more general use in controlling system performance.

The system configuration option `:fSuppressGlobalPolicy:` allows you to control the number of frames allocated to individual programs according to the partition size specified in the Context Manager configuration file, *CmConfig.sys*.

To use the option, you need to be aware of the working set requirements of each program. Allocating too few frames can cause local thrashing. Frequent disk accesses result from application pages competing for partition frames. Increasing in the frequency of disk activity can impede performance to a point of diminishing returns.

You might assert the `:fSuppressGlobalPolicy:` option, for instance, to maintain reasonable performance when an excessive memory consuming program is executing. To do this, specify a very large partition size for each program that does not use frames excessively and a smaller size for the frame consumer. This will contain the consumer in its own partition, causing it to thrash locally rather than impede the performance of other programs.

(For details on workstation configuration file options, see “Configuring Workstation Operating Systems,” in the *CTOS System Administration Guide*.)

Demand Paging Operations

The demand paging operation for application use is described below.
(See the *CTOS Procedural Interface Reference Manual* for a complete description of the operation.)

QueryPagingStatistics

Returns paging statistics for a user or for the paging service.

Section 4

Using CTOS Operations

Introduction to Using CTOS Operations

This section is provided to help you use the CTOS operations in the programs that you write. The contents are a collection of practical information, including examples of how to make calls to the operating system, an overview of the types of available system and library calls, ways to perceive and address memory, and levels at which you can access devices. Like other sections, it points out the advantages of using certain operations over others. For example, it touches upon the issue of standardization, an area of concern when developing future-oriented programs. In addition, the section points out the advantages of using memory in protected mode.

Before You Begin Programming on CTOS

It is assumed that the operating system has been successfully installed on your workstation and that you have installed the Standard Software Development Utilities. The Development Utilities include the Linker, the Librarian, the Assembler, and the standard operating system libraries (*Ctos.lib*, *CtosToolkit.lib*, and *Enls.lib*). (See your release documentation for more information.) In addition, you should have installed the language compiler for the high-level language you will be using.

If the above assumptions are correct, you can use your workstation for writing software programs.

You also should have available the documentation you will need to refer to while you are writing your programs. At a minimum, you will need the *CTOS Procedural Interface Reference Manual*. The *CTOS Programming Utilities Reference Manual: Building Applications*, the *CTOS Debugger User's Guide*, and the appropriate programming language manual are other supporting software manuals that you should have when you are ready to compile, link, and run your programs.

Naming Conventions

You will notice that certain conventions are used to name variables in the *CTOS Procedural Interface Reference Manual* and other supporting software manuals. You need to familiarize yourself with these conventions to understand what the variables mean when you write programs that use CTOS operations.

In the section entitled “About This Manual,” you are provided information on the naming conventions most commonly used. It is recommended that you follow these same conventions when developing your own software.

Programmatic Interface

The programmatic interface to any of the CTOS operations is a procedural call. The format of the procedural interface is given for each operation in the *CTOS Procedural Interface Reference Manual*. The following are examples of what this format looks like for three CTOS operations:

WildCardInit (pb, cb, pBuf, sBuf): ercType

PutFrameCharsAndAttrs(iFrame, iCol, iLine, pbText, cbText, pbAttrs, cbAttrs): ercType

OpenFile(pFhRet, pbFilespec, cbFileSpec, pbPassword, cbPassword, mode): ercType

The operation *name* is to the left of the left parenthesis. It is recommended that you capitalize letters consistently in this name, particularly if you are using a case sensitive compiler. Note how names are capitalized in the operation descriptions in the *CTOS Procedural Interface Reference Manual*.

The names enclosed within the parentheses are *variable names* representing *parameters*. Note that these variable names follow the naming conventions described in “About This Manual.”

For example,

pFhRet

means the memory address (*p*) of a location in your program into which a file handle (*Fh*) is to be returned (*Ret*).

The *CTOS Procedural Interface Reference Manual* includes a description for each operation. The description tells you what to fill in for each parameter to the procedural interface. (See “Example CTOS Call in C Language” for details.)

Almost all CTOS operations are written as *function calls* that return a one-word status code commonly known as an *erc*. Each of the example operations listed previously returns a status code and, therefore, is labeled *ercType*.

In many cases the status code indicates an error (and is often referred to as an error code) but this is not always true. In some cases, the value returned indicates the direction of program processing thereafter. For details on the meanings of returned codes, consult the *CTOS Status Codes Reference Manual* and the operation description in the *CTOS Procedural Interface Reference Manual*.

If an *ercType* operation returns a status code of 0, this is referred to as *ercOK* and means that the operation completed successfully. The operating system itself does not report any errors to the user; it simply returns status codes to programs that use operating system services. Programmers should always check the returned code and provide for error reporting or recovery.

Example CTOS Call in C Language

To use the procedural interface format, you must write it as a language statement. For example, the format of `OpenFile` looks like

```
OpenFile(pFhRet, pbFilespec, cbFileSpec, pbPassword, cbPassword,
mode): ercType
```

The following is an example of how you can fill in the parameters to `OpenFile` in C language. Each variable name (from left to right) is described and followed by what you write for it.

1. *pFhRet* is the address to which the file handle for the open file will be returned, for example:
&fh
2. *pbFileSpec* is the address of a file specification, for example:
&fileSpec
3. *cbFileSpec* is the length in bytes of the specification, for example:
sizeof(fileSpec)
4. *pbPassword* is the memory address of the file password. For example, no password required is indicated as
(char)0*
5. *cbPassword* is the length in bytes of the password. For example, no password is indicated as
0
6. *mode* is a two-letter constant indicating the mode in which the file is to be opened. For example, read mode is indicated as
0x6D72

The completed `OpenFile` statement in C is thus

```
erc = OpenFile(&fh, &fileSpec, sizeof(fileSpec), (char*)0, 0, 0x6D72);
```

Operation Types

Your program can use the procedural interface with any of the following types of operations:

- Object module procedures
- System-common procedures
- Operations that use the *request procedural interface* to system services
- Kernel primitives

Each CTOS operation in the *CTOS Procedural Interface Reference Manual* is identified as one of these types.

Each operation type functions in the operating system in a different way.

Object Module Procedures

An *object module* procedure is a procedure in a library. It is not part of the operating system code itself. The Linker can statically link an object module with your program as part of the code that is executed when your program is run.

Note: *The Linker also can dynamically bind procedures in dynamic link libraries (DLLs) to an application when the application is loaded. For details, see “Statically Linked Object Modules,” in the section entitled “Dynamic Link Libraries.”*

WildCardInit is an example of an object module procedure. When your program executes a call to WildCardInit, control is transferred to the WildCardInit code in your program. When WildCardInit has completed executing, it returns to the next executable instruction in your program.

WildCardInit is in the standard operating system library. All procedures in the system library are documented in “Operations” in the *CTOS Procedural Interface Reference Manual*.

System-Common Procedures

A *system-common procedure* does not reside in a library nor is it linked with your program. It is a procedure within the operating system itself. A system-common procedure is either so common that it should not have to be duplicated, or it is hardware-dependent code too extensive to be included in every program written.

Although there are system-common procedures built into the operating system, you can also write your own procedures and install them dynamically. A collection of related system-common procedures is called a *system-common service*. Like a request-based system service (see “System Service Processes”), a system-common service also serves clients. Unlike its request-based counterpart, however, a system-common service does not use request-based IPC messages for communication. Because there is no need for processes to wait at

exchanges for IPC messages, system-common services can increase program performance. On the other hand, such services can only serve locally based clients. (For a detailed comparison of system-common to request-based services, see the section entitled “System-Common Services Management.”)

PutFrameCharsAndAttrs is an example of a system-common procedure. This procedure is just one of several procedures collectively called the Video Access Method (VAM). VAM is a system-common service for programming to the video device. (For details on VAM, see “Using the Video Access Method (VAM)” in the section entitled “Video.”)

Note: Unless explicitly stated, the term system service refers to a request-based system service.

Kernel Primitives

The kernel primitives are part of the operating system. They are

Check	Request
ControlInterrupt	RequestDirect
CreateProcess	RequestRemote
DeviceInService	Respond
ForwardRequest	Send
PSend	Wait
	WaitLong

Using the Request Procedural Interface

The *request procedural interface* is a routine within the operating system used to access a system service. It calls the kernel primitive, Request, to do this. The request procedural interface is not linked with your application. Instead, an interrupt is generated, which transfers control to the request procedural interface routine. Your application is placed in a waiting state while the routine executes.

The request procedural interface first constructs a request block. The *request block* is a message used by all interprocess communications. It is constructed according to specific conventions from the parameters you supplied in the procedural interface.

The request procedural interface then calls `Request` to route the request block to the system service. When the system service completes its service, it fills in its response in the request block and calls the kernel primitive, `Respond`. `Respond` routes the request block back to your application.

Upon completion, a status code is returned to your application. A status code of 0 (`ercOK`) indicates that the system service performed the operation with no error.

The CTOS operations that use the request procedural interface are *request-based* operations. `OpenFile` is an example. You can identify the request-based operations in the *CTOS Procedural Interface Reference Manual* by the request block format following the operation description.

Using the Kernel Primitives

To access a system service using kernel primitives, you are required to construct the request block yourself for the specified request-based operation. Then you must call the kernel primitives, `Request` and `Wait` (or `Check`), for the request to be serviced.

This method of accessing a system service has the advantage of allowing your program to continue execution while it periodically checks for the response from the system service. The request procedural interface always requires that your program wait for the response. The request procedural interface, however, is easier to use.

It is recommended that you read the advanced sections in this manual before you use the kernel primitives in this way. (See the section entitled “Interprocess Communication,” for more information.)

I/O Interface Levels

Figure 4-1 shows the I/O sections in this manual. Each section (except “Input/Output,” which is introductory) presents the interfaces you can use to perform I/O to and from hardware devices.

I/O interfaces are available for the same device at different interface levels. The *level* of an interface implies the degree of control a program has over a hardware device when it uses that interface. *Low-level* interfaces provide greater hardware control than *high-level* interfaces but, at the same time, restrict a program to performing I/O to fewer devices.

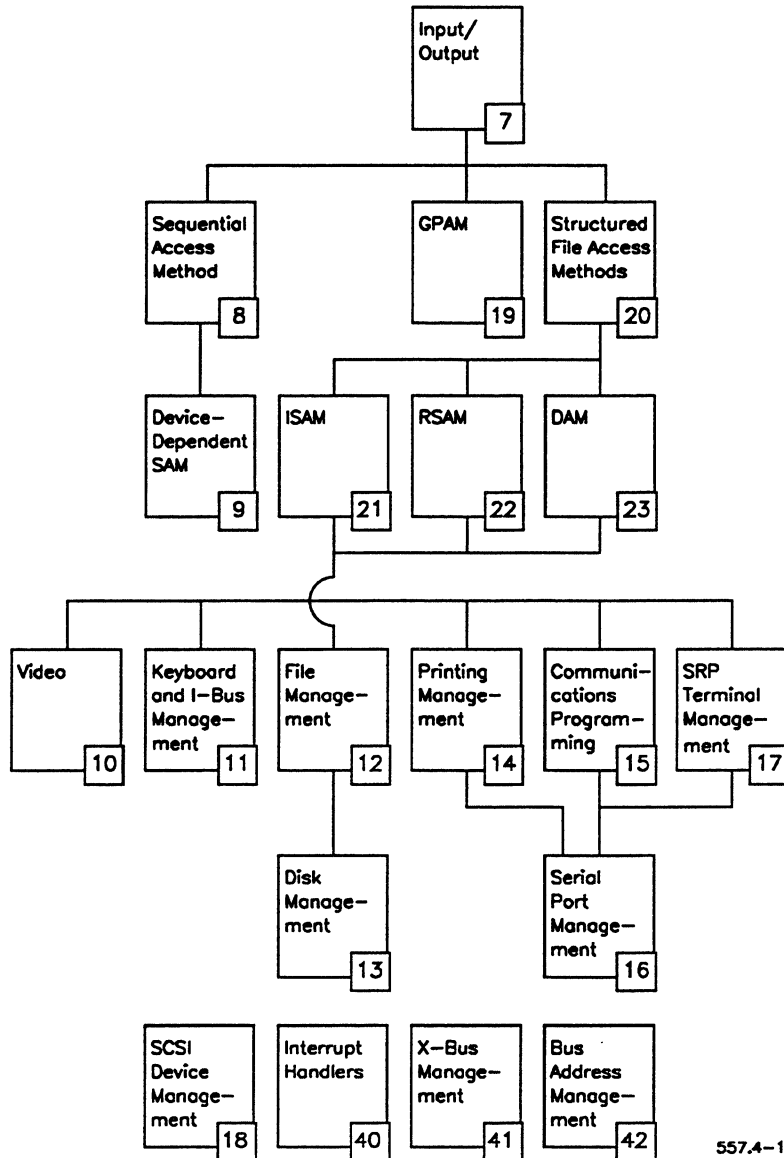
The sections closer to the top of Figure 4-1 describe high-level interfaces. Low-level interfaces are described in the sections towards the bottom. The sections with device names such as “Video” and “Disk Management,” for example, describe low-level interfaces.

If you are getting acquainted with CTOS operations, the easiest way to access a device is at a high level. For example, you can use the operations in the sequential access method (SAM) section to access the video device. The SAM interfaces are easier to use than the low-level video interfaces, because you write fewer statements in your program.

You will discover that there are advantages and disadvantages to using different interface levels.

The subject of interface levels is discussed at length in the I/O sections.

Figure 4-1. Interface Levels



557.4-1

Addressing Memory

A *physical memory address* (PA) is the actual location in system memory.

Real mode applications are capable of addressing up to 1 megabyte of physical address space. This means that a real mode application can reference each of the 1,048,576 bytes by a unique physical address.

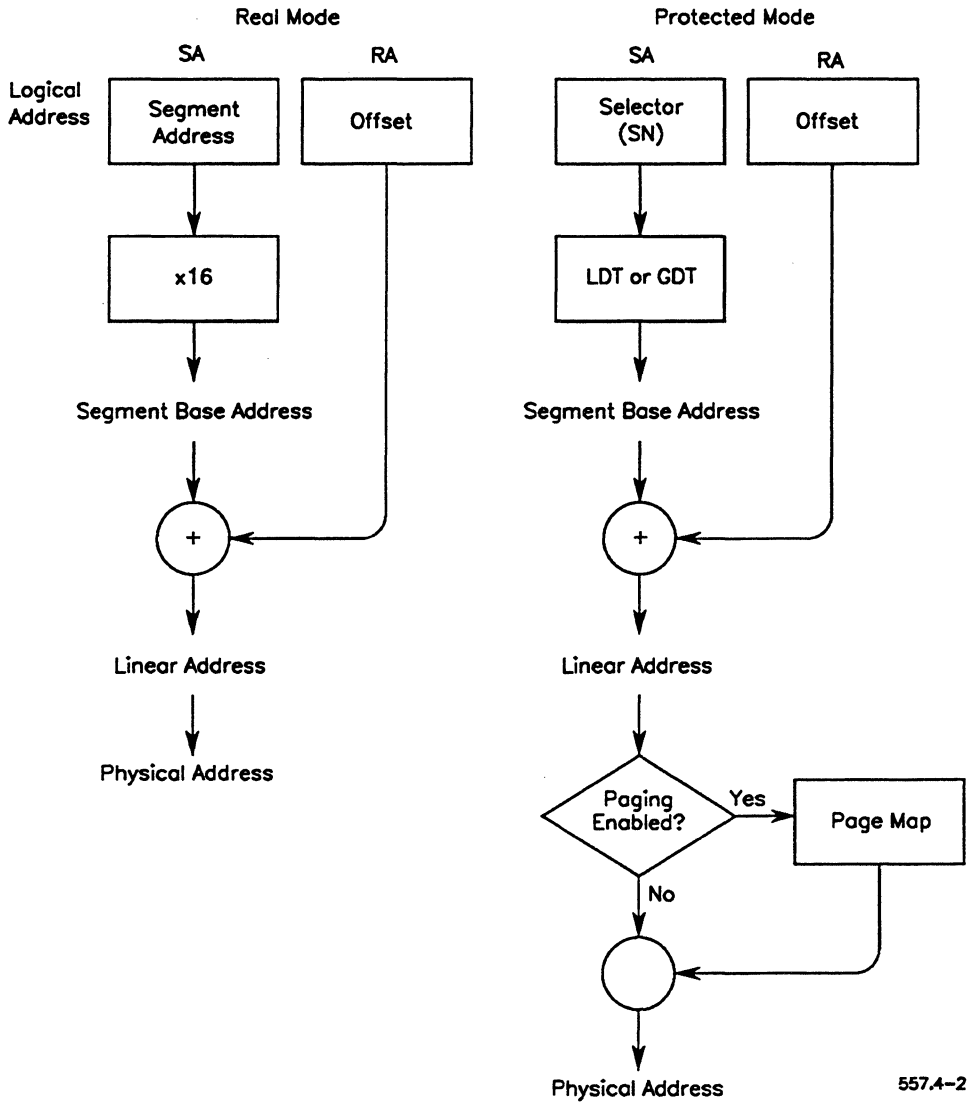
In protected mode, the physical address space extends beyond 1 megabyte. The amount of physical memory your application can address is determined by your system processor and its hardware limitations. An 80286 processor, for example, is capable of providing a 16 megabyte physical address space. The actual address space, however, is determined by the hardware.

(For details on protected mode addressing, see the Intel manuals listed in “Related Documentation,” at the beginning of this manual.)

A *segment* is a contiguous area in the *linear address* space. (See “Linear Memory Address.”) Typically, segments are less than 64K bytes. Applications can address *huge segments* greater than 64K bytes using 32-bit assembly language instructions. (For details on huge segments, see “Segments,” in the section entitled “Memory Management.”)

You can translate a memory address into three different forms. All operating systems support the translation of an address into its logical, linear, and physical address form. (See Figure 4-2.)

Figure 4-2. OS Concepts: Memory Address Translations



557.4-2

Logical Memory Address

The *logical memory address* is the memory address as viewed by an application program. For example,

pFh

is the logical memory address (denoted by p) of a file handle (denoted by Fh). Typically you use the logical translation of a memory address more frequently than its other forms.

Note: *Unless explicitly stated otherwise in the text, memory address means the logical memory address.*

The logical memory address consists of two parts: a *segment address* (SA) and a *relative address* (RA) commonly called the *offset*. The syntax of a logical memory address in assembly language is

SA:RA

The SA is the high-order portion of the logical address. Depending upon whether the processor is executing in real mode or in protected mode, the SA is interpreted differently, as described below:

- In real mode, the SA is multiplied by 16 to determine the *segment base address*.
- In protected mode, the SA is a *selector* (SN). It selects a segment descriptor entry in a protected mode system structure [either a *local descriptor table* (LDT) or a *global descriptor table* (GDT)].

The *segment descriptor* contains the segment base address, which may be located anywhere in memory. For this reason, if you are writing a program you intend to execute in protected mode, your program should not depend upon the value of the SN. (For details on writing protected mode programs, see the *CTOS Programming Guide*.)

The RA or offset is the low-order portion of the logical address. It is the number of bytes from the beginning of the segment to the target location.

A byte of memory does not have a unique logical memory address. The same byte of memory can be referred to by many different combinations of SAs and RAs.

Linear Memory Address

The *linear memory address* is the memory address as viewed by the processor. The linear address space is the range of addressable memory the processor provides (tempored by hardware limitations on variable partition and virtual partition operating systems). Linear addresses are expressed relative to address 0 and can be compared to each other on this basis.

The linear address translation is computed by adding the offset to the segment base address. The linear address is equivalent to the physical memory address on multipartition and variable partition operating systems. (See Figure 4-2.) On virtual memory operating systems, linear addresses are mapped to physical addresses by means of paging structures. The linear address space on virtual memory systems is determined by the processor only. It is not affected by the hardware.

Physical Memory Address

The *physical memory address* is the actual location in RAM.

- In real mode on multipartition and variable partition operating systems, the physical address is equivalent to the linear address.
- In protected mode on variable partition operating systems, the physical address is equivalent to the linear address.
- On virtual memory systems, the linear addresses of protected mode and real mode applications are mapped to the physical address by means of a page map structure. (Real mode applications have a unique mapping to physical memory by means of a local page map structure. For details, see “Executing Real Mode Applications,” in the section entitled “Demand Paging.”)

Advantages to Protected Mode Memory Addressing

On variable partition operating systems, protected mode memory addressing provides two significant advantages. It allows programs to use extended memory beyond the first megabyte of physical memory, and it protects the memory allocated to one program from being referenced by other programs.

Extended Memory

Protected mode extends memory, allowing you to run programs beyond the first megabyte of physical memory. Real mode applications, however, are limited to 1 megabyte of addressability.

As an end user, this means you can run more applications in memory. As a programmer, you can reference physical memory addresses extending beyond the first megabyte up to the maximum allowed by your processor and hardware.

Protection

In protected mode, programs are prevented from referencing static memory allocated to other programs, or from overwriting code. This is because LDTs and GDTs provide limit and type checking, which place limitations on the memory programs can access.

Selecting Operations for Program Portability

In “Introduction to CTOS,” you were introduced to nationalization as a feature offered by all CTOS operating systems. If you are just beginning to write programs to be run on CTOS, you should be sure your application can easily be ported among operating systems customized to reflect different native languages. To avoid having to change application code, you should observe the following guidelines:

- Select the nationalized version of an operation whenever there is a choice. In most cases, nationalizable operation names can be identified by the prefix *Enls* or *Nls*. (For details, see the section entitled “Native Language Support.”)
- If there is an ENLS equivalent for an NLS operation, use the ENLS operation. (For details, see “Native Language Support.”)
- Not all nationalizable operations are identified by the prefix *Enls* or *Nls*. Two exceptions are FormEdit and MenuEdit. (See the section entitled “Utility Operations.”) Operations that are exceptions to the naming standard are identified as they are introduced.
- Use separate message files for the messages a program displays. Separate message files can be customized to the requirements of a language at any time without changing the program code. (For details, see “Native Language Support.”)
- To process file specifications, use the parsing and building file specification operations. (For details on these operations, see the section entitled “File Management.”)

To achieve the greatest flexibility in creating portable programs, you should always link your programs with the appropriate standard operating system libraries. (See your release documentation for details.) Also, when selecting from the operations described in this manual, you should check Appendix J, “Operation Data,” in the *CTOS Procedural Interface Reference Manual* to be sure that the operation is supported on your system. (You can do this programmatically by a call to either the `CurrentOsVersion` or `OsVersion` operation. See the *CTOS Procedural Interface Reference Manual* for details.)

Section 5

Program Management

What Is Program Management?

This section describes what constitutes a program, how a program is loaded into memory, and what happens when that program terminates. In addition, the operations available to handle runtime error conditions are described. In “Partitions and Partition Management,” you are introduced to additional program operations that enable a partition managing program to manage other programs executing in memory at the same time.

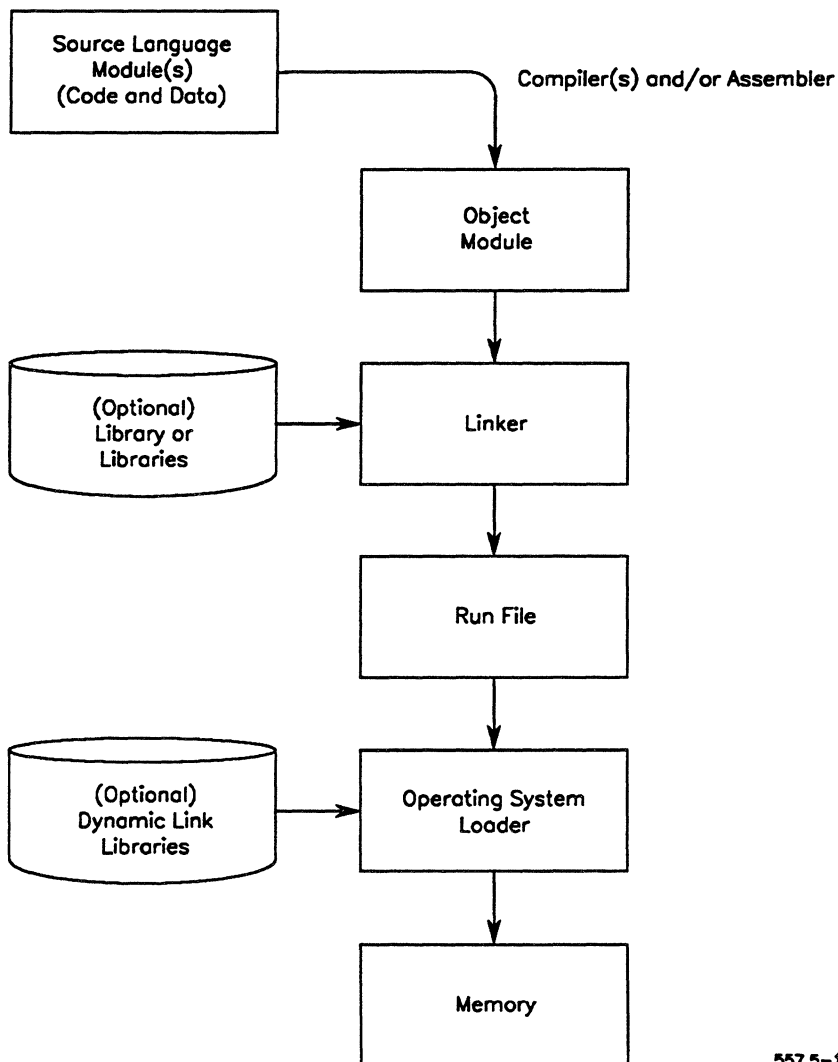
In this section you are introduced to the term partition. The real significance of partitions (*or user numbers*) and partition structures, however, will become more evident in the context of the multiprogramming environment described in “Partitions and Partition Management.” For now, you can think of a *partition* as a block of memory into which a program is loaded. The primary focus at the moment is not on multiprogramming, but a single program and the operations available to it.

What Is a Program?

An executable *program* can consist of code, data, and one or more processes in a partition in memory.

A program is loaded into memory from a disk-resident file called a *run file*. Run files are created by compiling and/or assembling source language modules into object modules and linking the object modules together into code and data segments. (See Figure 5-1.)

Figure 5-1. From Source Modules to Program in Memory



557.5-1

Linking a Program

When you link your program, the Linker reads the object module(s) you previously compiled or assembled. To create the executable run file, the Linker combines the *segment elements* (such as code, data, and constants contained within the modules) according to their segment names, class names, and user-provided directives.

On protected mode operating systems, you can size a program at link time. Sizing means controlling both of the following:

- The maximum amount of memory the program can allocate for data
- The minimum amount of memory that the operating system will allocate for the program before attempting to run it

(For details on the Linker, see the *CTOS Programming Utilities Reference Manual: Building Applications*.)

The resulting run file created by the Linker consists of *segments*. *Code segments* contain only processor instructions (code) or read only data and are never modified once they are loaded into memory. *Static data segments* contain initial values of program data structures and are writable once in memory.

The Linker can group segments based on one of several models of segmentation. Although small and large model are used, most programming languages use the medium model of segmentation. (See your programming documentation for details on segmentation models. See the section entitled “Stack Format and Calling Conventions” in *CTOS/Open Programming Practices and Standards* for a discussion of the medium model.) A medium model run file created by linking object modules produced by the Pascal compiler, for example, consists of one code segment for each object module included in the link and a single static data segment. The single static data segment, or DGroup, combines the static data and stack requirements of all the object modules.

Such a run file is considered standard; assembly language programmers are urged to adopt this standard unless other considerations are overriding. The COBOL compiler and BASIC interpreter do not produce object modules.

Because every memory address is relative to a segment, the subject of segments will be taken up again from the viewpoint of memory addressing in the section entitled “Memory Management.” That section also describes how data segments can be allocated dynamically (while a program is executing).

Loading a Program Into Memory

A program can be loaded into memory by the Chain, Exit, or ErrorExit operation.

When a program is loaded, the run file is read into the short-lived memory of the application partition. For real mode programs, any logical memory addresses existing in either the code or data segments (intersegment references) are adjusted to reflect the memory address at which the program is loaded. For protected mode programs, the Loader adjusts the base addresses in each Local Descriptor Table (LDT) descriptor to reflect the memory address at which the program is loaded.

On multipartition and variable partition operating systems, the virtual code management facility allows you to run a program that is larger than the available memory in an application partition. When in use, virtual code management loads all the static data segments and the resident code segment into memory. The programmer controls which nonresident code segments are to be placed in overlays and loaded as needed. (For details, see “Virtual Code Management.”)

On virtual partition operating systems, the virtual code management facility is not needed. Applications that use overlays, however, can run without change. The overlays are simply treated like ordinary program pages: they are faulted into physical memory frames by the paging service when the code they contain is requested.

Note: *The video byte stream output of a program loaded into memory by the Chain operation can be redirected to a file. (For details, see “Using a Byte Stream” in the section entitled “Sequential Access Method.”)*

Program Termination

The application program can terminate itself by using the Chain, Exit, ErrorExit, or ExitAndRemove operation.

Loading the Exit Run File

When the currently executing program terminates, the *exit run file* is the next program that is loaded into the partition. Exit run files are user-specified. Each application partition has its own. For example, the Executive sets itself as the exit run file: the user starts the application from the Executive, and when the application is done, the Executive is reloaded. Applications that run under partition managing programs also have exit run files. With Context Manager, for example, the exit run file for such an application requests that Context Manager do everything that is necessary to remove the terminating partition.

A program (not under partition management) can specify an exit run file for its partition using the SetExitRunFile operation. Furthermore, the program can determine the exit run file of its partition using QueryExitRunFile.

If no exit run file is specified in a partition, the partition becomes vacant.

Notifying Other Programs of Termination

When a program terminates, the operating system issues termination requests. *Termination requests* (system requests) are messages that notify system services of a program's termination. Upon receipt of a termination request, system services release resources, such as open files, that may be allocated to the terminating program. Termination requests are described in detail in "System Services Management."

Deallocating System Resources

Only the resources allocated to a program are deallocated when that program terminates.

Among the resources deallocated are the following. For details on each, see the appropriate section reference.

- Short-lived memory. (See “Memory Management.”)
- Exchanges. (See “Interprocess Communication.”)
- Files opened by the OpenFile operation (except long-lived files). (See “File Management.”)
- Timer request blocks allocated by the OpenRTCClock operation. (See “Timer Management.”)
- Communications channels allocated by the InitCommLine operation. (See “Serial Port Management.”)
- Global descriptor table selectors (SGs) (protected mode). (See the Intel manuals listed in “Related Documentation,” at the beginning of this manual.)

Error Handling

The operating system makes available a number of error handling operations that you can use to test for error conditions and, if necessary, to terminate your program. To determine the error code returned by an operating system procedure, for example, your program can call the CheckErc operation. CheckErc is particularly useful for debugging runtime errors. (For details, see the *CTOS Debugger User's Guide*.) To terminate a program, the ErrorExit operation can be called. For details and examples of how to use CheckErc and ErrorExit, see the *CTOS Programming Guide*.

Program Management Operations

The program management operations described below are categorized as error handling and normal program exit operations. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Error Handling

CheckErc

Checks status codes. If *CheckErc* is called with a nonzero status code, *FatalError* is called with that value.

FatalError

Terminates operation of the application program and passes an abnormal status code to the exit run file.

ErrorExit*

Terminates the current application program in an application partition and passes an abnormal status code to the exit run file.

ErrorExitString*

Terminates the current application program in an application partition and returns a string (usually printed) to the exit run file. If the exit run file is the Executive, the string is displayed to the video device.

Crash

Causes system operation on a workstation to terminate, a crash dump to be written, the operating system to be reloaded, and *SignOn* to display the cause of the crash when it is restarted. On a shared resource processor, *Crash* works differently. (For details, see the *CTOS Status Codes Reference Manual*.)

*Dynamically installed system services use these operations at a certain time during installation. (For details, see "System Services Management.")

SetMsgRet

Is the same as `ErrorExitString` except the program does not exit.

Normal Program Exit

Exit*

Terminates the current application program in an application partition and passes a normal status code to the exit run file. To do this, `Exit` calls the `ErrorExit` operation with `ercOK`. (As such, your program can call `ErrorExit` directly with `ercOK` rather than calling `Exit`.)

Chain*

Replaces the current application program in an application partition with the specified run file.

ExitAndRemove

Terminates the current application program and removes the specified vacant partition. The user number is deallocated and becomes available to be reissued.

SetExitRunFile

Establishes a new exit run file for an application partition.

QueryExitRunFile

Returns the name, password, and priority of the exit run file of an application partition.

*Dynamically installed system services use these operations at a certain time during installation. (For details, see "System Services Management.")

Section 6

Parameter Management

What is Parameter Management?

The parameter management facility provides a structured mechanism for passing limited information from one application program to its successor within the same application partition.

Program Using Parameter Management

The Executive is a typical example of an application program that uses the parameter management facility.

The Executive interfaces with the user through a forms-oriented interface. A *forms-oriented interface* accepts parameters from the user.

The Executive thus passes user-supplied parameters to other programs. The way that the Executive does this is described below. (See the *CTOS Executive Reference Manual* for details.)

In the Executive, the user types a command name on the command line. When the user presses **RETURN**, the Executive is given the command.

The Executive responds by writing the user-requested command form to the screen. The command form contains the appropriate prompts for the user to enter data.

If the user, for example, types **Rename** on the command line and presses **RETURN**, the following command form appears:

Command Form

Rename

File from

File to

[Overwrite OK?]

[Confirm each?]

The command form consists of a list of prompts. The user enters data on the lines (parameter fields) in the form next to the prompts, correcting typing errors if necessary. When satisfied with the contents of the fields, the user presses GO to execute the command.

In the example, the Executive would pass the parameters to the Rename program. The Rename program, in turn, renames the user-specified files.

A forms-oriented interface, such as the Executive, is one type of program that can use the parameter management facility to its advantage. Parameter management, however, can be used by any application program in a partition that needs to provide information to any other program that will run in the same partition.

Parameters

A *parameter* consists of zero or more subparameters.

In the Executive **Rename** command described earlier, for example, the prompt [Confirm each?] accepts either of the following:

- Zero parameters (meaning the user did not enter any information)
- One parameter (a Yes answer)

A *subparameter* typically consists of an arbitrary sequence of characters not including a space.

The prompt, File from, in the Executive **Rename** command allows the user to enter one or more file names. Each file name is a *subparameter*; the *parameter* is the complete file list the user entered on the File from line.

As another example, the parameter

1 abc Work.Fri

contains three subparameters, which are 1, abc, and Work.Fri. The space is the delimiter that separates the subparameters.

A space can be embedded within a subparameter by including the entire subparameter in single quotes. For example, the parameter

'1 abc' Work.Fri

contains two subparameters: 1 abc and Work.Fri.

Structures and Operations Overview

Programs using the parameter management facility must organize parameter data to simplify the method in which other programs extract the parameters.

The organized data is stored in the *variable length parameter block* (VLPB), a data structure in long-lived memory of the application partition. The memory address of the VLPB is stored in the *application system control block* (ASCB) of the partition.

To place parameter data in an organized fashion into the VLPB, programs can use the parameter management operations for constructing the VLPB. (These operations are described in "Operations for Constructing the VLPB.")

To extract parameters from the VLPB, programs can use the parameter management operations for querying the parameters stored in that structure. (These operations are described in "Querying Parameters in the VLPB.")

Application System Control Block (ASCB)

An *application system control block* (ASCB) is automatically created in an application partition when the partition is created. The ASCB contains the memory addresses of various types of partition-specific information, such as the VLPB. This information is available to be queried by programs, such as the Executive, which execute in the partition. (See the section entitled "System Definitions," for details on how a program can obtain partition information from the ASCB. For details on the ASCB structure, see "Application System Control Block" in "System Structures," in the *CTOS Procedural Interface Reference Manual*.)

Variable Length Parameter Block (VLPB)

The *variable length parameter block* (VLPB) is a partition structure used by the parameter management facility to communicate parameters to programs.

The VLPB is created in the long-lived memory of an application partition. Its memory address is stored in the *pVLPB* field of the ASCB.

Conceptually, the VLPB can be described as a two-dimensional sparse array of strings. The Executive command form illustrates the parts of this array as follows:

- Each *element* (iParam, jParam) in the array is the value of a subparameter entered into an Executive command form.
- Each *row* (iParam) of the array corresponds to a line in the command form. The first row corresponds to the command name. Each subsequent row corresponds to a parameter.
- Each *column* (jParam) of the array corresponds to a subparameter.

Querying Parameters in the VLPB

A program can query the variable length parameter block (VLPB) to obtain parameter information by using three operations: RgParam, CParams, and CSubParams.

- RgParam returns the memory address of the array element specified by (iParam, jParam). Each element of the array returned by RgParam is actually a 6 byte block of memory called an *sdType*. The first 4 bytes are the memory address of the string. The last 2 bytes are the length of the string.
- CParams returns the number of parameters stored in the VLPB. CParams, for example, is the number of parameter fields displayed in an Executive command form.
- CSubParams returns the number of subparameters stored in the VLPB for a specified parameter. CSubParams, for example, is the number of subparameters the user entered in a specified field of an Executive command form.

Figure 6-1 shows the matrix of a VLPB array for the Executive.

The Executive places the following information in row 0 (iParam 0):

- The Executive *command name*, such as Rename, is placed into element (0,0).
- The case value entered when the command was created is placed into element (0,1). The *case value* specifies which command invoked the current *run file* (disk resident file) when more than one possibility exists. The case value can be queried by a run file to determine which command invoked it.
- The redo keystroke buffer is placed into element (0,2). The *redo keystroke buffer* contains the entire series of keystrokes typed from the time the user entered the command name up to but not including the keystroke GO that executed the command.

Rows 1 through n store the parameters and subparameters that the user entered in the command form.

Figure 6-1. Executive Variable Length Parameter Block

rgParams (VLPB)	SubParam (iParam) 0	SubParam (iParam) 1	SubParam ... (iParam) ... 2 ...	SubParam (iParam) n
Param 0 (iParam 0)	<command name>	<case>	<Redo keystroke buffer>	
		.		
		.		
		.		
Param n* (iParam n)	(n,0)	(n,1)	(n,2)	
*where the values in row <u>n</u> are the subparameters of the nth parameter				

557.6-1

Example of a VLPB for the Rename Command

If the Executive **Rename** command were filled out as shown below, the variable length parameter block (VLPB) would look like the matrix shown in Figure 6-2.

Command Form

Rename	
File from	abc def
File to	g hi
[Overwrite OK?]	y
[Confirm each?]	y

When the user presses **GO**, the Executive organizes the data to simplify the extraction of the parameters.

The **RgParam** operation provides access to the parameters by returning to the caller the memory address of the array element specified by (iParam, jParam).

In Figure 6-2, for example, the memory address of abc is returned by **RgParam (1,0)**; the address of def is returned by **RgParam (1,1)**.

Figure 6-2. Filled-in Variable Length Parameter Block

rgParams (VLPB)	SubParam (jParam) 0	SubParam (jParam) 1	SubParam (jParam) 2
Param (iParam) 0	Rename	00	Rename abc def g hi y y
Param (iParam) 1	abc	def	
Param (iParam) 2	g	hi	
Param (iParam) 3	y		
Param (iParam) 4	y		

557.6-2

Operations for Constructing the VLPB

The following operation sequence is recommended to initialize a variable length parameter block (VLPB):

- Call `ResetMemoryLL` to reset the long-lived memory of the partition. Note that `ResetMemoryLL` also deletes the contents of the Redo keystroke buffer.
- Call `AllocMemoryLL` to allocate the number of bytes required for containing the VLPB structure.
- Call `RgParamInit` to initialize the specified memory for the VLPB.

The construction of parameters for a VLPB is supported by three object module procedures: `RgParamSetSimple`, `RgParamSetEltNext`, and `RgParamSetListStart`.

`RgParamSetSimple` creates one subparameter per row of the VLPB sparse array.

To construct a VLPB array with more than one subparameter per row, a program must first call `RgParamSetListStart`. `RgParamSetListStart` sets the global variable for placing the subparameters in the VLPB. Following a call to `RgParamSetListStart`, a call to `RgParamSetEltNext` must be made for each subparameter to be created in the row.

The VLPB and the parameter-passing services of the Executive are applicable to any application program using the operating system.

VLPB Structure

The variable length parameter block (VLPB) structure is a self-describing, two-dimensional array of character strings. Each element of the array `rgSdoParam` is a pair (`ob`, `cb`) of words, where

- `ob` is the offset within the VLPB of the corresponding row of the two-dimensional array
- `cb` is the number of bytes occupied by the row

The strings that make up a row are prefixed with a 1 byte count and packed together without padding.

When a program uses the operations for constructing a VLPB, the VLPB structure is filled in with values.

(See “Variable Length Parameter Block,” in the *CTOS Procedural Interface Reference Manual*, for the format of the VLPB.)

Parameter Management Operations

The parameter management operations described below are categorized by function. (See the CTOS Procedural Interface Reference Manual for a complete description of each operation.)

Constructing Parameters

The operations below are used by only a few systems programs to construct parameters.

RgParamInit

Initializes the specified memory to be the VLPB.

RgParamSetEltNext

Creates an additional subparameter of the current parameter in the VLPB.

RgParamSetListStart

Initiates the creation of a parameter with multiple subparameters.

RgParamSetSimple

Creates a parameter with one subparameter.

Querying Parameters

The operations below are used by every program to query parameters in the VLPB.

CParams

Returns the number of parameters stored in the VLPB.

CSubParams

Returns the number of subparameters stored in the VLPB for a specified parameter.

RgParam

Provides access to the parameters stored in the VLPB.

Section 7

Input/Output

In This Section

This section is a guide to the operating system I/O facilities. It presents interface options available and discusses considerations you need to make regarding these options.

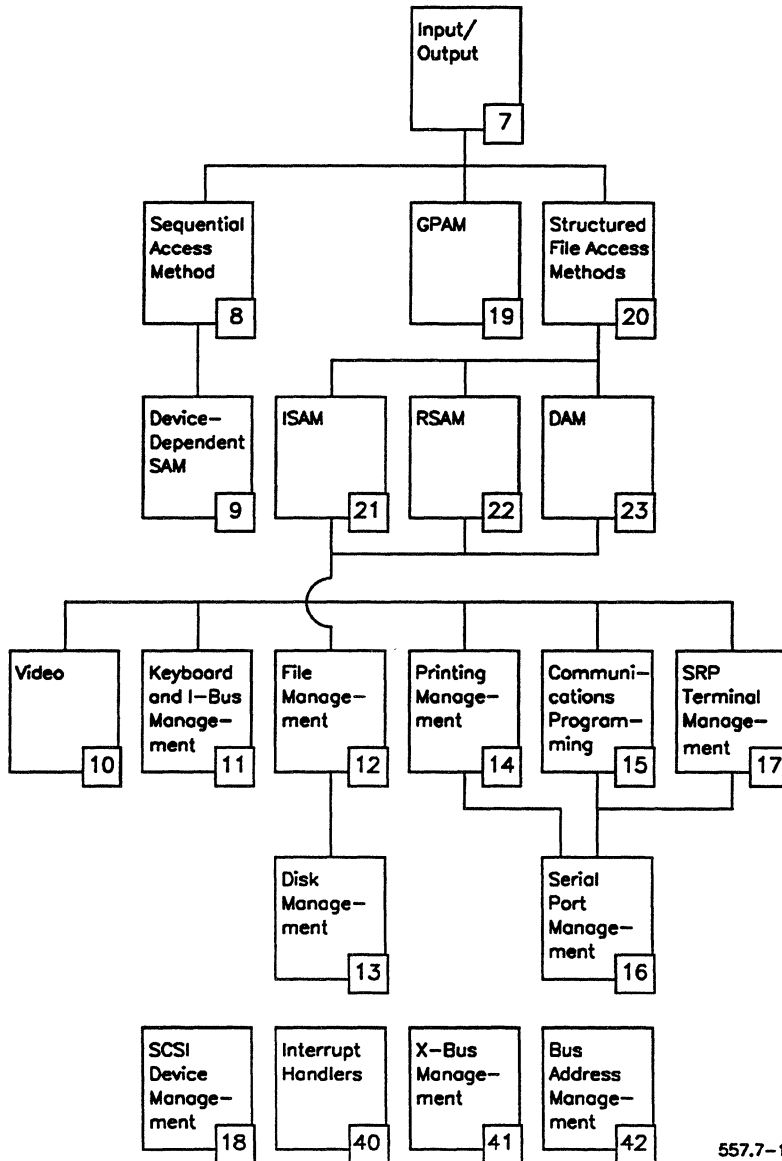
Manual Sections Describing I/O

Figure 7-1 shows the I/O sections in the *CTOS Operating System Concepts Manual*. Each section (except this section, which is introductory) presents interfaces you can use to send I/O to a hardware device.

I/O interfaces are available for the same device at different interface levels. The *level* of an interface implies the relative degree of program control over a hardware device. *Low-level* interfaces provide greater hardware control than high-level interfaces but, at the same time, limit program I/O to fewer devices.

In Figure 7-1, the sections towards the top describe *high-level* interfaces. Low-level interfaces are described in the sections towards the bottom.

Figure 7-1. Interface Levels



557.7-1

Device Independence Versus Dependence

A program's capability to run on various devices is a characteristic built into the program's design as a result of its interface level.

A *device-independent* program is capable of performing I/O to devices of different types, such as video, tape, or keyboard. Using high-level interfaces, thus, results in device independence.

You can write a device-independent program, for example, by using the Sequential Access Method (SAM) operations.

A *device-dependent* program performs I/O to a limited number of devices or a particular type of device. Using the low-level interfaces, thus, results in device-dependence.

The video device, for example, has its own group of video operations for performing I/O functions. These operations result in video device-dependence. The file system, keyboard, and other devices have their own comparable operations.

A device-dependent program provides for more direct control over the physical device but, at the same time, requires more effort to write.

I/O Facilities

Any of the following I/O facilities can be used for the same device (see Figure 7-1):

- High-level to low-level device access to data:

Sequential Access Method (SAM). SAM is known more familiarly as byte streams. Using these high-level interfaces results in device-independence. (See the section entitled "Sequential Access Method.")

Device-Dependent SAM. Using certain operations at this lower level results in device-dependence. (See the section entitled "Device-Dependent SAM.")

Device level. The operations at this level are specific to a given type of device and thus result in device-dependence. (The device-named sections such as “Video” and “Keyboard and I-Bus Management” are shown towards the bottom of Figure 7-1.)

The sections entitled “Disk Management” and “Serial Port Management” describe operations that are even closer to the actual device details than the operations described in the other device-named sections.

SCSI devices, interrupt handlers, X-Bus, and I/O buffer management. The sections entitled “SCSI Device Management,” “Interrupt Handlers,” “X-Bus Management,” and “Bus Address Management” describe operations associated with more than one device.

- **High-level device access to special kinds of data:**

Generic Print Access Method (GPAM). GPAM provides high-level I/O for complex documents that may include text, graphics, or special text attributes. GPAM is an object module library that provides device independent formatting commands used for printing. (See the section entitled “Generic Print Access Method.”)

- **High-level device access to structured data files:**

File access methods. The section entitled “Structured File Access Methods” is a guide to three high-level I/O interfaces to structured data files.

Section 8

Sequential Access Method

What is the Sequential Access Method?

SAM is a set of interfaces that provide device-independent access to a default set of real devices, such as the screen, printer, files, and keyboard. To transfer data to or from the device, SAM uses a character-oriented sequence of bytes known as a *byte stream*. SAM consists of object module procedures in the standard operating system library.

SAM is an alternative to the direct programming interfaces available at the device-dependent level. (The device-dependent interfaces are listed in sections such as “Video” or “Serial Port Management,” which are associated with device names.)

With SAM you can write a program that can be

- Used flexibly to access any of the available devices
- Written with a minimum amount of code

If, for example, you want to write a compiler program that accepts its data from either the keyboard or a file and directs its listing to the screen, a printer, or a file, it would be to your advantage to use SAM’s device-independent level of interface.

If, however, you know that your program will always perform I/O to a single device, it would be to your advantage to use the device-dependent level of interfaces for that device.

Device-dependent interfaces are specific to each kind of peripheral device available on a workstation. Programming at the device-dependent level has the advantages of

- Maximizing run time efficiency
- Providing access to the specialized features of the peripheral device hardware (for example, controlling the cursor at the video level)

Customizing the Sequential Access Method

The default devices that SAM supports are as follows:

- Disk
- Parallel printer
- Spooler
- Keyboard
- Null
- Video

For some applications, you may not need to use all of the devices supported by SAM. For example, a program might use SAM only to obtain keyboard input and to display text on the screen.

If this is the only way you use SAM in a particular application, you can configure SAM's device-dependent object modules selectively to support only the devices you need.

You generate SAM (*SAMGen*) by editing an assembly language file containing byte stream definition strings, assembling the file, and linking the resulting object module with your program.

Specific uses of a *SAMGen* are listed below:

- Reduction of the memory needed by an application program by eliminating unneeded device support
- Inclusion of support for communications and RS-232-C serial communications printers
- Inclusion of support for tape
- Inclusion of support for the Generic Print System (GPS)

- Inclusion of user-written, device-specific SAM object modules

(For details on customizing SAM object modules, see the *CTOS Programming Guide*.)

Byte Stream

A *byte stream* is a readable (input) or writable (output) sequence of 8-bit bytes. An *input byte stream* can be read until either the reader chooses to stop reading or until status code 1 (“End of file”) is returned. An *output byte stream* can be written until the writer chooses to stop writing. (Of course there are physical limitations: a file could expand, for example, to fill all available disk storage.)

A *byte stream work area* (BSWA) is a memory work area for the exclusive use of SAM operations. Any number of byte streams can be open concurrently, using separate BSWAs. An application must allocate a BSWA for each byte stream it opens. (For details on the BSWA, see the *CTOS Programming Guide*.)

Note: *The video byte stream and keyboard byte stream are preopened and can be accessed by referring to the external array variables bsVid and bsKbd, respectively.*

Byte streams use a caller-supplied buffer for all buffering operations. The caller must provide a separate buffer for each BSWA it allocates. The buffer is not required, however, for video or keyboard byte streams.

Using a Byte Stream

To open a byte stream, the `OpenByteStream` operation is called with the following parameters:

- The device/file specification string from the list in “Device/File Specifications”
- A password if appropriate
- The mode (type of I/O needed, such as read, write, or both)
- The address of the BSWA
- The address and size of the user-allocated buffer

When calling other device-independent operations such as `ReadBsRecord`, `WriteBsRecord`, or `CloseByteStream`, you supply the address of the same BSWA.

There are two predefined and already allocated BSWAs (*bsVid* for video frame 0 and *bsKbd* for the keyboard). With video byte streams, your program must first have established video frame 0 using VAM or the Executive. (See the section entitled “Video,” for details.)

These special BSWAs are defined in SAM standard object modules. Because these BSWAs are already opened, it is not necessary (nor allowed) to specify them as arguments to `OpenByteStream` or `CloseByteStream` (with the one exception noted in the next paragraph). The byte streams may be used by passing the memory address of *bsVid* or *bsKbd* to the appropriate byte stream operations.

In one special case, a program must explicitly call `OpenByteStream`, specifying *bsVid* for video frame 0. This occurs when the video byte stream output of a program is to be filtered to a file rather than to be displayed to the video device. Filtering of the program’s video output is controlled by a controlling program. The controlling program uses the two object module procedures, `OpenVidFilter` and `CloseVidFilter`, to initiate and to terminate filtering of the byte stream.

After calling the `OpenVidFilter` operation, the controlling program sets itself as the exit run file. Then it calls the `Chain` operation to chain to the controlled program. When the controlled program terminates, the controlling program calls `CloseVidFilter`. In this special case, the caller-allocated buffer to `OpenByteStream` is used to redirect the byte stream to the file.

Types of Byte Streams

The types of byte streams that SAM supports are

- Disk byte streams
- Printer byte streams
- Generic Print System byte streams
- Pre-GPS spooler byte streams
- Keyboard byte streams

- Communications byte streams
- X.25 byte streams
- Video byte streams
- Sequential access byte streams

Disk Byte Streams

A *disk byte stream* is a byte stream that uses a file on disk. A valid file name follows the standard file naming conventions.

Disk byte streams permit both input and output to be directed to the same open byte stream (that is, the same BSWA).

The standard operations of SAM are augmented by two operations that allow random access to files: `GetBsLfa` and `SetBsLfa`. These device-dependent operations are available only for disk byte streams and return status code 7 ("Not implemented") if attempted on other byte streams. (For details, see the section entitled "Device-Dependent SAM.")

(For a comparison of using disk byte streams to using the file management operations to write to a file, see "Performing I/O" in the section entitled "File Management.")

Printer Byte Streams

A *printer byte stream* is a byte stream that performs direct printing. Valid strings for printer byte streams are `[LPT]` and `[PTR]n`. *n* is any valid RS-232-C serial communications channel in a `[COMM]` device specification if a printer is attached to that serial port. (For details on communications channels, see "Device/File Specifications.")

Direct printing transfers text directly from application program memory to the specified parallel or serial printer interface of the workstation on which the application program is executing. A printer byte stream cannot be used to access a printer assigned to the GPS or to the spooler byte stream. (See "Generic Print System Byte Streams" and "Pre-GPS Spooler Byte Streams.")

The selected configuration file determines the printer characteristics. (See the **Create Configuration File** command in the *CTOS Executive Reference Manual*.) For example, the configuration file controls whether a printer byte stream suspends execution of the caller until the workstation operator corrects a condition requiring manual intervention or reports it to the calling program.

Normally printer byte streams change tab and end-of-line characters to the form expected by the printer. Return (code 0Ah), for example, can be transformed to a Carriage Return/Linefeed combination for some printers, or just to a Carriage Return (code 0Dh) or to a Linefeed (code 0Ah) for others. Tab characters can be transformed to spaces for printers without mechanical tabs. These transformations are controlled by the selected configuration file.

Any of three printing modes can be specified with the `SetImageMode` operation: normal, image, or binary. `SetImageMode` sets the printing mode any time following the opening of the printer byte stream. This differs from the effect of `SetImageMode` when used with pre-GPS spooler byte streams.

For compatibility between spooled and direct printing, `SetImageMode` should be used before the first `WriteBsRecord` or `WriteByte` operation.

Normal mode converts tabs into spaces and converts end-of-line characters to device-dependent codes.

Image mode and *binary mode* perform no code conversion.

Binary mode does not print the banner page or send any extra code not in the file to the printer, nor does it recognize the escape sequences. *Escape sequences* are special character sequences that invoke special functions.

Generic Print System Byte Streams

A *Generic Print System (GPS) byte stream* is a byte stream that is sent to a GPS printing device. GPS byte streams supersede pre-GPS spooler byte streams. (See “Pre-GPS Spooler Byte Streams.” Also see “Device/File Specification Parsing.”)

For compatibility with pre-GPS spooler byte streams, GPS byte streams implement the `SetImageMode` operation in the same way as pre-GPS spooler byte streams.

Pre-GPS Spooler Byte Streams

(For details on pre-GPS printing, see the *CTOS Programming Guide*.)

A *pre-GPS spooler byte stream* automatically creates a uniquely named disk file for temporary text storage. It then transfers the text to the disk file and expands the disk file as necessary. When the spooler byte stream is closed, a request is queued for the spooler by the Queue Manager for later printing of the previously created disk file. The temporary file is deleted after it is printed. This is *spooled printing*.

Normally, pre-GPS spooler byte streams change tab and end-of-line characters to the form expected by the printer. For example, a system Return (code 0Ah) can be transformed to a Carriage Return/ Linefeed combination for some printers, or just to a Carriage Return (code 0Dh) or a Linefeed (code 0Ah) for others. Tab characters can be transformed to spaces for printers without mechanical tabs. These transformations are controlled by the selected configuration file. (For details, see the **Create Configuration File** command in the *CTOS Executive Reference Manual*.)

Any of three printing modes can be set with the `SetImageMode` operation: normal, image, or binary. `SetImageMode` sets the printing mode only if it is called immediately following the opening of the spooler byte stream. This differs from the effect of `SetImageMode` when used with printer byte streams. (See “Printer Byte Streams.”)

For compatibility between spooled and direct printing, `SetImageMode` should be used before the first `WriteBsRecord` or `WriteByte` operation.

Normal mode prints the banner page between files, converts tabs into spaces, converts end-of-line characters to device-dependent codes, and recognizes the escape sequences for manual intervention. (For details on banner pages, see the *Printing Guide*.)

Image mode prints the banner page between files and recognizes the escape sequences, but performs no code conversion.

Binary mode does not print the banner or send any extra code not in the file to the printer, nor does it recognize the escape sequences.

Keyboard Byte Streams

A *keyboard byte stream* is equivalent to using the ReadKbdInfo or ReadKbd operation in *character mode*. (For details on keyboard program modes, see “Keyboard and I-Bus Management.”) The keyboard byte stream does not support unencoded keyboard mode.

To support device-independence, keyboard byte streams return status code 1 (“End of file”) when the **FINISH** (ASCII value 4) key is pressed, and status code 4 (“Operator intervention”) when the **CANCEL** (ASCII value 7) key is pressed.

Communications Byte Streams

A *communications byte stream* is a byte stream that uses an RS-232-C serial communications channel (serial port). Communications byte streams provide support for the two communications channels of the serial input/output (SIO) communications controller. Operation is in asynchronous, full-duplex mode without explicit modem control. Like disk byte streams, communications byte streams permit both input and output to be directed to the same open byte stream (that is, the same BSWA). Only one byte stream can be opened for each communications channel of the SIO controller.

The selected configuration file determines the communications characteristics. (For details, see the **Create Configuration File** command in the *CTOS Executive Reference Manual*.)

Normally, communications byte streams strip null (00h) and delete (7Fh) characters from the stream of received data characters. Image mode (set with the SetImageMode operation) specifies that communications byte streams pass all incoming characters to the requesting program exactly as received.

X.25 Byte Streams

An *X.25 byte stream* is a byte stream that enables data transmission by means of the X.25 Network Gateway.

Each open X.25 byte stream corresponds to a virtual circuit that is initiated when the byte stream is opened, and cleared when the byte stream is closed. Setting up and clearing of the virtual circuit is controlled through the use of a configuration file.

Note: *X.25 byte streams are not packaged with Standard Software. They are separately orderable.*

Video Byte Streams

A *video byte stream* is a byte stream that uses the video display. The standard SAM operations are augmented by the following:

- Certain characters that have special interpretation.
- Multibyte escape sequences. The multibyte escape sequences (beginning with the character 0FFh) can be used to control the special workstation video capabilities.
- One device-dependent operation. The QueryVidBs operation returns information about video byte streams.

(See the section entitled “Video,” for details on video byte streams and on other ways to control the video subsystem.)

Sequential Access Byte Streams

A *sequential access* byte stream (also known as a *tape byte stream*) reads or writes a sequential access device in a strictly sequential fashion. It looks for the pattern of file marks that designates the beginning and end of a file. The Sequential Access Service must be installed before sequential access byte streams can be used.

Note: *Sequential access byte streams are also called tape byte streams. Currently supported sequential access devices include quarter-inch cartridge (QIC) tape, half-inch tape, and 4mm digital data storage (DDS) tape.*

With sequential access byte streams, you can read or write to a sequential access device using the standard byte stream interface. When opening a sequential access byte stream in read mode, records are read from the device as a sequence of bytes until a file mark is encountered. The user is not aware of the record size. Write mode is used to write data to the sequential access device.

Sequential access byte streams are not included in the standard SamGen. You must create and use a customized SamGen. (For details, see the section entitled “Building a Customized SAM” in *CTOS/Open Programming Practices and Standards*.)

Sequential access byte streams use a configuration file. The configuration file name must be in the format *XXXconfig.sys*, where *XXX* is the name of the sequential access device (for example, QIC or TAPE) specified in the call to *OpenByteStream*. The sequential access device configuration file is created using the **Configure Sequential Access Device** command. (All other byte streams use the **Create Configuration File** command. See the *CTOS Executive Reference Manual* for command details.)

For optimal performance when performing sequential access I/O, it is recommended that you use the default values for the sequential access device configuration file parameters *TotalServiceBuffers* and *ServiceBufferSize*. Only in the case where the data to be written requires more than one volume may it be necessary to specify values for these parameters.

Device/File Specifications

The device/file specification string is any of the following:

[Node][VolName]<DirName>FileName

File identified by its full file specification. Abbreviated specifications are also allowed.

[LPT]&[VolName]<DirName>FileName

Centronics-compatible printer connected to the parallel printer port.

&[VolName]<DirName>FileName is optional. It describes a configuration file containing the printer characteristics. A default configuration file is used if none is specified. (For details, see the **Create Configuration File** command in the *CTOS Executive Reference Manual*.)

[PTR]n&[VolName]<DirName>FileName

RS-232-C-compatible printer, where n identifies the serial I/O (SIO) communications channel to which the printer is connected and can be any of the channels listed below.

&[VolName]<DirName>FileName is optional. It describes a configuration file containing the printer characteristics. A default configuration file is used if none is specified. (For details, see the **Create Configuration File** command in the *CTOS Executive Reference Manual*.)

[COMM]n&[VolName]<DirName>FileName

Communications channel n of the SIO communications controller, where n identifies the channel.

&[VolName]<DirName>FileName is optional. It describes a configuration file containing the communications characteristics. A default configuration file is used if none is specified. (For details, see the **Create Configuration File** command in the *CTOS Executive Reference Manual*.)

Valid channel identifiers are listed below:

Channel Synonyms	Processor Channel	Device
A 0 0A	A	Workstations, shared resource processor (SRP) - TP, CP, GP, GP+SI, and GP+CI
B 1 0B	B	Workstations, SRP-TP, CP, GP, GP+SI, and GP+CI
C 2	C	SRP - TP, CP, GP+CI
D 3	D	SRP - TP and GP+CI
E 4	E	SRP - TP and GP+CI
F 5	F	SRP - TP and GP+CI
G 6	G	SRP - TP and GP+CI
H 7	H	SRP - TP and GP+CI
I 8	I	SRP - TP only
J 9	J	SRP - TP only

Sequential Access Method

On a GP+CI, channels E and F are configurable for RS-232-C or V.35; channels G and H are configurable for RS-232-C or X.21. (For details on shared resource processor features, see "Operating System Types.")

The following specifications support the communications expander, for example, the XC-002 port expander module:

1A	Leftmost XC-002, Channel A
1B	Leftmost XC-002, Channel B
1C	Leftmost XC-002, Channel C
1D	Leftmost XC-002, Channel D
2A	Second XC-002, Channel A
2B	Second XC-002, Channel B
2C	Second XC-002, Channel C
2D	Second XC-002, Channel D

Note: *The above channels are valid only when the XC-002 system service is installed.*

[Node][SeqAccessDeviceName]n&[VolName]<DirName>FileName

Sequential Access Service. The sequential access device name should match the name specified when the Sequential Access Service is installed. Examples of sequential access device names are [QIC], [TAPE], [DDS], and [!QIC].

The optional node name specifies the node on which the Sequential Access Service is installed, if the service is not installed locally.

n specifies the file position for reading the sequential access device. When reading from a sequential access device, legal file position formats are [XXX] and [XXX]*n*, where [XXX]0 denotes the same file as [XXX]. When writing, the only legal file position formats are [XXX], [XXX]0, and [XXX]+. (This last format causes the new file to be appended at the end of data on the medium.)

The sequential access device files are read using the sequential numbering scheme: 0, 1, 2, and so on. For example, a file stored in the second QIC tape position can be read by specifying a name such as [QIC]1.

[Node][QueueName]ReportName

Spooled printer. The queue name is the name of the pre-GPS scheduling queue associated with the spooler. [SPL] is the default name of the first spooled printer.

The report name is a text string of up to 12 characters that is included in the Spooler Status command's status display. (For details, see the *CTOS Executive Reference Manual*.)

[KBD]

Keyboard. This also includes the system input process used for submit files and batch jobs. (For details on the system input process, see "System Input Process" in the section entitled "Keyboard and I-Bus Management." Batch is described in the *CTOS Batch Manager II Installation, Configuration, and Programming Guide*.)

[X25]n&[VolName]<DirName>FileName

X.25 virtual circuit, where n is a network identification that currently must be zero.

&[VolName]<DirName>FileName is optional. It describes a configuration file containing the circuit characteristics.

[NUL]

Null device. Input operations always return status code 1 ("End of file"). Output operations discard all output but return status code 0 (ercOK).

[VID]

Video frame 0. The frame must be established in advance using the Video Access Method (VAM) or the Executive.

[VID]n

Video frame n.

Device/File Specification Parsing

To determine the type of byte stream you are specifying, SAM parses the device/file specification string supplied to `OpenByteStream`. This *string parsing* process is described below.

Scanning from left to right, SAM first looks for a left bracket ([).

If a left bracket ([) is not found and disk byte streams are included in the SAM configuration, SAM assumes the string to be a file name. The byte stream is a disk byte stream, which is directed to a disk file.

If a left bracket ([) is found, SAM attempts to match the string characters and the string length within the square brackets to the reserved words for system devices, such as KBD, LPT, and PTR.

1. If a match occurs, SAM specifically looks for any characters to the right of the right square bracket (]).
 - a. If a left angle bracket (<) is found, the string is assumed to be a file name, and the byte stream is therefore a disk byte stream.
 - b. If no characters are found, the string is a reserved word for a device, and the device byte stream is directed to the specified device.
2. If no match occurs and GPS is installed, SAM assumes the byte stream is a GPS byte stream. Otherwise, if the spooler is installed, the byte stream is assumed to be a pre-GPS spooler byte stream.

SAM Operations

The SAM operations described below are categorized as basic or advanced. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Basic

OpenByteStream

Opens a device/file as a byte stream.

ReadBsRecord

Reads the specified count of bytes from the open input byte stream to the specified memory area.

ReadByte

Reads 1 byte from the open input byte stream.

WriteBsRecord

Writes the specified count of bytes to the open output byte stream from the specified memory area.

WriteByte

Writes 1 byte to the open output byte stream.

CloseByteStream

Closes the open byte stream.

OutputToVid0

Provides programs, such as system services, with the ability to perform minimal output to the video device without linking to a full video byte stream.

GetBsMember

Returns the contents of the specified BSWA field.

Advanced

ReadBytes

Reads up to the specified count of bytes from the open input byte stream. **ReadBytes** returns the memory address of the start of the byte stream but does not move the bytes to a separate buffer.

CheckpointBs

Writes any partially full buffers of the open output byte stream and waits for all write operations to complete successfully before returning.

ReleaseByteStream

Abnormally closes the device/file associated with the open output byte stream.

QueryVidBs

Allows your program to obtain information about a video byte stream.

Section 9

Device Dependent SAM

What is Device-Dependent SAM?

The sequential access method (SAM) highlights the device-independent aspect of SAM. If you use the basic operations described in the section entitled “Sequential Access Method,” you allow your program to be portable to a number of devices.

SAM, however, has a device-dependent portion to its code for each type of device it supports.

Mapping to Device-Dependent Operations

The device-independent operations map to device-dependent operations, which are specific to each device. Mapping is done automatically each time a device-independent operation is called. It is based on information stored in the Byte Stream Work Area. (See “Byte Stream” in the section entitled “Sequential Access Method.”)

Calling a device-independent operation results in mapping to a device-dependent operation with a generic prefix.

To send output to an open line printer byte stream, for example, you would call the device-independent operation, `WriteBsRecord`. `WriteBsRecord`, in turn, calls the device-dependent operation, `FlushBufferLp`. The generic prefix is `FlushBuffer`. `LP` (the name of the specific device) is appended to the prefix.

The device-independent operations and the generic prefixes to their device-dependent versions are as follows:

Device-Independent Operation	Generic Prefix
OpenByteStream	OpenByteStream...
ReadByte, ReadBsRecord	FillBuffer...
WriteByte, WriteBsRecord	FlushBuffer...
part of CloseByteStream	CheckPointBs...
part of CloseByteStream	ReleaseByteStream...
SetImageMode	SetImageMode...

(For details, see the *CTOS Programming Guide*.)

Device-Specific Operations

To handle select types of byte streams in special ways, you can incorporate certain device-specific operations directly into your program. The following are device-specific operations:

Operation	Applicable Byte Streams
SetImageMode	Communications, printer, Generic Print System (GPS), pre-GPS spooler
PutBackByte	Disk (async and sync)
GetBsLfa	Disk (async and sync)
SetBsLfa	Disk (async and sync)
QueryVidBs	Video

If you use these operations, you limit your program to specific devices. If, for example, you use GetBsLfa in your byte stream, your program will work only if you specify a disk file name.

Note that, although `GetBsLfa` and `SetBsLfa` pertain to files, these operations are called only through byte streams and are therefore included in this section rather than in the section entitled “File Management.” The same is true of `QueryVidBs`, which is included here instead of in the section entitled “Video.” `QueryVidBs` is a byte stream path for manipulating the video device.

(See the *CTOS Programming Guide* for details on how to use these operations in customizing your program.)

Device-Dependent SAM Operations

The device-dependent SAM operations described below are categorized by interface function. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Operations With Generic Prefixes

Every type of byte stream has operations whose names *begin* with the prefixes below.

OpenByteStream...

Opens a specific device/file as a byte stream.

FillBuffer...

Reads data from the device into a user-specified buffer.

FlushBuffer...

Writes data from a user-specified buffer to the device.

CheckPointBs...

Ensures that all data in the buffer has been output to the device (forms part of the *CloseByteStream* operation).

ReleaseByteStream...

Releases the device for use by other programs (completes the *CloseByteStream* operation).

SetImageMode...

Affects the interpretation of bytes read from or written to the device (for example, controls whether tabs are expanded or not).

Device-Specific Operations

The operations below limit your program to specific devices.

SetImageMode

Sets the normal, image, or binary mode for printer, spooler, and communications byte streams.

PutBackByte

Returns 1 byte to the open input disk byte stream.

GetBsLfa

Returns the logical file address at which the next I/O operation will occur for the open disk byte stream.

SetBsLfa

Sets the logical file address at which the I/O operation is to continue for the open disk byte stream.

QueryVidBs

Returns video information about the type of video device associated with an open video byte stream.

OpenVidFilter

Sets the flag *fVidFilter* in the application system control block. The flag redirects to the temporary file *[Scr]<\$>bsFilter.tmp* the video byte stream output of a program to which the caller chains.

DeleteByteStream

Deletes the file associated with the open byte stream identified by the memory address of the byte stream work area.

CloseVidFilter

Clears the flag *fVidFilter* in the application system control block that redirected video byte stream output to the temporary file *[Scr]<\$>bsFilter.tmp*. *CloseVidFilter* also closes the open byte stream if the program whose output was being filtered had not already done so.

RenameByteStream

Changes the file name and/or the directory name of the file associated with the open byte stream identified by the memory address of the byte stream work area.

Section 10

Video

What is the Video Facility?

This section describes the video facility. The video facility is described from the following viewpoints:

- How you can use it to your advantage in your programs
- What video capabilities are available to you with each hardware type

For details on programming using color, see the *CTOS Programming Guide*.

The video facility is a highly flexible means for the display of alphanumeric and graphic information. Workstation video is of two types: character-map and bit-map.

Character-Map and Bit-Map Video

Although most *character-map* workstations can be equipped to display graphics, the primary feature is the video hardware contained to support the character map. The hardware reads characters and attributes from memory. It then converts them from the extended ASCII (8 bit) memory representation to a pattern of illuminated dots, called *pixels*, that it displays on the screen. During this conversion, the video hardware references a translation table (font) that is loaded into the video hardware under program control. Character-map fonts are created with the Font Designer.

A *bit-map* workstation does not contain hardware to support the character map (although it contains graphics hardware). Instead, the video software provides character-map emulation to support character-only application programs. The font can be modified but is of a different format than the character-map font. Bit-map fonts are created with the Raster Font and Icon Designer.

Video Attributes

Video attributes can be either screen or character attributes and control the visual presentation of characters on the screen.

- *Screen attributes* control the presentation of the entire screen. Examples are blank, reverse video (dark characters on a light background), half-bright, number of characters per line, and the presence or absence of character attributes.
- *Character attributes* control the presentation of a single character. Examples are reverse video, blinking, half-bright, underlining, bold, and struck-through.

Note: *There are certain limitations to normal video capabilities when using standard VGA on EISA/ISA-bus workstations. Manipulating video attributes programmatically, for example, may not result in the display of those attributes or the display may be different than expected. More specifically, blinking has limitations. The underlining, bold, and struck-through attributes are not supported. Through system configuration options, however, color can be substituted for these attributes. For details on standard VGA, see the software release announcement for VAM 4.0.*

Video Software

The video software consists of a device-independent and a device-dependent level of interface to the video facility. Each level provides varying degrees of screen and character attribute control.

The screen consists of a number of separate, rectangular areas called *frames*. Each frame can be scrolled up or down independently of other frames. You can select from several features, including multiple frames and scrolling of each frame, to enhance your program video output.

The video software consists of the following two interface levels:

- At the device-independent level, you can use the sequential access method (SAM). SAM provides device-independent access to devices such as the printer, files, keyboard, as well as the screen. (See the section entitled “Sequential Access Method.”) SAM provides automatic scrolling. Video-specific extensions to the SAM provide direct cursor addressing, control of character attributes, and so on.
- At the device-dependent level, you can use either of the following:
 - The Video Access Method (VAM). VAM operations provide you with direct access to the characters and character attributes of each frame. They include explicit control of scrolling.
 - The video display management facility (VDM). VDM consists of operations for controlling screen attributes. For example, the VDM operations enable you to split the screen into frames. VDM and VAM can be used together or independently.

Although VAM and VDM operations direct program output only to a video device, they allow writing to the video on any type of workstation.

Using the SAM Operations

To direct your program output to the screen, you can use the SAM device-independent operations through the current screen setup or by making explicit program calls.

Using the Current Screen Setup

If you are writing a program such as a utility that will be invoked by the Executive to display messages in a streaming or sequential way, you do not need to initialize the video display. Instead, you can take advantage of the Executive screen setup. Screen setup allows you to use the device-independent SAM operations, such as `OpenByteStream`, specifying the video [VID] as your device string. SAM then generates a *video byte stream* for use by the video display. You can alternately use the pre-opened byte stream, `bsVid`.

The Executive eliminates the need to reinitialize the video because your program, when invoked, inherits the Executive

- Character font
- Character map (in system memory)
- Three frames (command frame, event frame, and status frame)

which comprise the Executive's *current screen* setup.

The SAM video byte stream extensions support multiple frames, character attributes, and explicit positioning of characters in a frame, but do not support line attributes (other than cursor position). SAM recognizes a few special cursor-positioning characters including **RETURN**, **NEXT PAGE**, **BACK SPACE**, and **TAB**. When a special character or full line would cause the cursor to move below the bottom line of the frame, SAM automatically scrolls the frame and repositions the cursor.

Using SAM Directly

If you choose not to have your program use the Executive screen setup, you can still use the device-independent SAM operations as previously described, but you also must initialize the screen. (See "Using Video Display Management (VDM).") If, for example, you want your program to be invoked directly by Context Manager but you don't expect Context Manager to set up the Executive screen, you must use VDM to initialize the screen.

Augmenting the SAM Operations

If you want greater control over the video byte stream, you can augment the SAM device-independent operations by the following:

- Special interpretation of certain characters. (See "Special Characters in Video Byte Streams.")
- Multibyte escape sequences. The multibyte escape sequences (beginning with the character 0FFh) can be used to control the special video capabilities of a workstation. (See "Multibyte Escape Sequences.")
- One device-dependent operation. The operation QueryVidBs returns information about video byte streams. (See "Using QueryVidBs.")

Special Characters in Video Byte Streams

(See Table H-7 in the *CTOS Procedural Interface Reference Manual* for the special characters interpreted by video byte streams.) Note that a multibyte escape sequence is available to disable all these special interpretations except 0FFh.

Multibyte Escape Sequences

Multibyte escape sequences can be used to perform the following functions:

- Control screen attributes
- Control character attributes
- Control scrolling and cursor positioning
- Dynamically redirect a video byte stream
- Automatically pause between full frames of text
- Various other miscellaneous functions

Note that where the escape sequences include alphabetic characters, uppercase and lowercase are equivalent.

Controlling Screen Attributes

Screen attributes can be controlled with four multibyte escape sequences. (See Table H-4 in the *CTOS Procedural Interface Reference Manual*.) Each of the 3 byte sequences begins with the escape byte 0FFh and continues with a pair of characters represented by the specified 8 bit ASCII character codes.

Controlling Character Attributes

Character attributes can also be controlled with multibyte escape sequences. (See Table H-2 in the *CTOS Procedural Interface Reference Manual*.)

Workstations support six character attributes: blinking, bold, half-bright, reverse video, struck-through, and underline.

You can use the escape sequence for subsequent characters in video byte streams to set all six character attributes in any combination.

Controlling Scrolling and Cursor Positioning

Characters are normally written to the frame sequentially, with the cursor advancing one character position at a time, from left to right and top to bottom. A cursor is normally displayed at the character position where the next character will be displayed. Text is automatically scrolled each time a character is written to the lower right corner of a frame. When such a scroll occurs, the entire contents of the frame scroll up one line, and the contents of the previous top line of the frame disappear.

(See Table H-5 in the *CTOS Procedural Interface Reference Manual* for the escape sequences that directly control scrolling and cursor positioning.)

Dynamically Redirecting a Video Byte Stream

When a video byte stream is opened, it is designated as directed to one of the frames. However, a special escape sequence makes it possible to dynamically redirect a video byte stream.

An independent cursor position is recorded for each frame. The position within frame *i* is restored automatically when a video byte stream is redirected to frame *i*. (See Table H-1 in the *CTOS Procedural Interface Reference Manual*.)

Automatically Pausing Between Full Frames

Automatic pausing between full frames of text can be controlled through multibyte escape sequences. When this pause facility is enabled and further output to the frame would cause text to be scrolled off the top of the frame, the following message is displayed on the last line of the frame:

Press **NEXT PAGE** or **SCROLL UP** to continue

At this point, if the user presses **NEXT PAGE**, output continues for another full frame of text. If the user presses **CANCEL**, status code 4 (“Operator intervention”) is returned to the calling process. If the user presses **FINISH**, status code 1 (“End of file”) is returned to the calling process. If the user presses any other key, the audio alarm is momentarily activated. (See Table H-3 in the *CTOS Procedural Interface Reference Manual* for the escape sequences controlling pause.)

Since the automatic pause facility reads characters from the keyboard (using the `ReadKbdInfo` operation with bit 5 of the *mode* parameter set or using `ReadKbdDirect`), there is potential for interaction with the client’s use of the keyboard.

A single client using a keyboard byte stream and one or more video byte streams will operate correctly. A more complex environment may require using program-specific logic to control pauses in scrolling. Automatic pausing can be affected by the following:

- Use of the unencoded keyboard mode
- Use of `ReadKbdInfo` or `ReadKbd` instead of a keyboard byte stream
- Keyboard input performed by one client while another uses a video byte stream
- Keyboard input initiated by the kernel primitive, `Request`, but not immediately followed by the kernel primitive, `Wait`

Miscellaneous Functions

See Table H-6 in the *CTOS Procedural Interface Reference Manual* for a description of the escape sequences that perform miscellaneous functions.

Using QueryVidBs

The QueryVidBs operation returns information about a video byte stream, such as frame number or current line number. (See the description of QueryVidBs in “Operations” in the *CTOS Procedural Interface Reference Manual*.)

Using the Video Access Method (VAM)

If you want more direct control over the screen than SAM provides, you can use the *Video Access Method* (VAM) operations. If your program does not require special screen setup, you can use the VAM operations independently of the video display management (VDM) operations.

VAM provides direct access to the characters and character attributes of each frame. VAM operations can be used to perform the following functions:

- Put a string of characters anywhere in a frame.
- Specify character attributes for a string of characters.
- Scroll a frame up or down a specified number of lines.
- Position a cursor in a frame. (Each frame can have its own cursor.)

Using Video Display Management (VDM)

If you choose not to use the Executive's screen setup or if your program is not invoked by the Executive, you can reinitialize the video subsystem using the video display management (VDM) operations before using the VAM or SAM operations.

The VDM facility sets up the screen. By using the VDM operations, your program can do the following:

- Determine the video capability present.
- Load a new character font into the font RAM.
- Stop video refresh on a character-map workstation (useful when moving or changing the size of the frames or the character map).

- Change screen attributes, such as reverse video and half-bright, while the screen is being video-refreshed.
- Calculate the amount of memory needed for the character-map based on the preferred height and width of the characters, and the presence or absence of character attributes.
- Initialize each of the frames.
- Initialize the character map.

Once the character map is set up and video refresh is started, you can use the VAM or the SAM operations to control the screen image by modifying the characters and attributes stored in the character map.

Reinitializing the Video Subsystem

Your program must reinitialize the video display if the intended state is not the same as that provided by the Executive.

To reinitialize the video display, you would include a sequence of VDM operations similar to the following:

1. Use the QueryVidHdw operation to determine the level of video capability present on the workstation in use.
2. Optionally use the LoadFontRam operation to read the character font from a file to memory and then load this font into the font RAM.
3. Use the ResetVideo operation to place the following information in the Video Control Block:
 - Number of characters per line
 - Number of lines per screen
 - The presence or absence of character attributes
4. Use the InitVidFrame operation to specify the screen coordinates and dimensions of each of the frames.
5. Use the SetScreenVidAttr operation to set reverse video or half-bright, if wanted.
6. Use the InitCharMap operation to initialize the character map.
7. Use the SetScreenVidAttr operation to initiate video refresh.

On bit-map workstations, you do not have to turn video refresh off and on during initialization.

On character-map workstations that have graphics capability, use the **SetScreenVidAttr** operation to turn off video refresh. **SetScreenVidAttr** turns off only the characters, not the graphics. However, on bit-map workstations where graphics and characters are not separated, both are turned off.

Following reinitialization, your program can display information by using VAM or SAM.

The Executive also allows you to use the **Screen Setup** command to respecify the following video characteristics:

- Reverse video
- Number of characters per line
- Number of lines
- Presence or absence of character attributes
- Suppressing pause between pages
- Color
- Screen timeout

(For details on the **Screen Setup** command, see the *CTOS Executive Reference Manual*.)

Forms-Oriented Interaction

VAM is ideal for forms-oriented interaction, that is, interaction in which a form is displayed in a frame and the workstation user enters data into the blank fields of the form. Direct cursor addressing and modification of individual characters and character attributes support this interaction.

For example, the **PutFrameAttrs** operation is used to highlight the field to be entered next. It sets reverse video for the range of character positions that constitute the field. After the field is entered, **PutFrameAttrs** is used again to reset the reverse video attribute on the character positions of the field.

Advanced Text Processing

VAM is also ideal for applications that perform advanced text processing, because it provides scrolling up and down of entire or partial frames. It is easy, for example, to scroll up the top four lines of a frame and insert a new line of text between the old fourth and fifth lines. During scrolling, character attributes scroll along with the text they affect.

Workstation Video Capabilities

The workstation types and models have different video capabilities. These are summarized in Table 10-1 and Table 10-2. (See the *CTOS System Administration Guide* for information on configuring the video for your workstation.) Table 10-1 shows workstation capabilities for all workstations except the B25 and B27. B25 and B27 capabilities are shown in Table 10-2.

In the discussion below, the descriptions of video capabilities apply to either character-map or bit-map workstations, unless specified otherwise.

Table 10-1. Workstation Video Capabilities

	Character Map		Standard VGA	Bit-map		
	(1) NonEV	EV		Low-Res	Hi-Res	Hi-Res Zoomed
Character Attributes						
Blinking	Yes	Yes	Yes (2)	(3)	(3)	(3)
Bold	Yes	Yes	No (4)	Yes	Yes	Yes
Half-bright	Yes	Yes	Yes	(5)	Yes	No
Reverse video	Yes	Yes	Yes	Yes	Yes	Yes
Struck-through	Yes	Yes	No (4)	Yes	Yes	Yes
Underline	Yes	Yes	No (4)	Yes	Yes	Yes
Loadable font	Yes	Yes	Yes	Yes	Yes	Yes
Characters/line	80	80 or 132	80	80	80	146
Lines/screen	29	29 or 34	29	29	38	38

Notes

1. NonEV means a character-map workstation without Enhanced Video (EV) capability.
2. Blinking is supported with standard VGA but the display is limited.
3. Blinking is substituted with an outline character.
4. Not supported with standard VGA but can be substituted with color.
5. Half-bright is emulated for consistency across the hardware, but it is recommended that you do not use it in your programs for the low-resolution monitor.

Table 10-2. B24 and B27 Workstation Video Capabilities

	B24*	B27
Character Attributes		
Blinking	Yes	Yes
Bold	No	Yes
Half-bright	Yes	Yes
Reverse video	Yes	Yes
Struck-through	No	Yes
Underline	Yes	Yes
Italic	No	No
Low contrast	No	No
Loadable font	Yes	Yes
Characters/line	40 or 80	80 or 132
Lines/screen	19 or 30	30 or 34
Double high, double wide	Yes**	Yes
Blinking cursor	Yes	Yes
Block cursor	No	Yes

*Cluster workstations only.

**With a 19 character/line x 40 line/screen definition.

Character Cell

Table 10-3 shows the character cell sizes available for character-map and bit-map workstations.

Table 10-3. Character Cell Size

Workstation Type Model*	Size
Character-map (nonEV)	9 x 12
Character-map (EV) and B27	7 x 10 7 x 12 9 x 10 9 x 12
EISA/ISA-bus standard VGA	8 x 16
Bit-map Low-resolution monitor	9 x 12
High-resolution monitor	12 x 20
High-resolution zoomed monitor	7 x 20

*For B24 workstations, character cell size is 8 x 10 in 80 column mode and 16 x 16 in 40 column mode.

Based on the character cell size of your workstation, you can obtain other information describing the level of video capability programmatically by using the QueryVidHdw or the QueryVideo operation. (For details, see the descriptions of these operations the *CTOS Procedural Interface Reference Manual*.)

Character-map

On a character-map workstation, characters displayed on the screen are stored in a contiguous area called the *character map*. The video controller has its own RAM containing a 4K byte character map and a 4K byte soft font.

The character map consists of 2K bytes of words. Each word in the character map contains one ASCII character byte (low byte) and one attribute byte (high byte) that applies only to that specific character. The map and font can be updated at any time and the result is immediately visible on the screen.

On a bit-map workstation, there is no video controller with its own character map: the character map is a software virtual map.

Video Attributes

Screen attributes control the presentation of the entire screen. The screen attributes are blank, half-bright, and reverse video.

Character attributes control the presentation of a single character. Character attributes can be present or absent, depending on the value of a screen attribute. If character attributes are present, then each character has an 8 bit character attribute field; 6 of the 8 bits in the character attribute field are used to specify the presence or absence of the attributes: blinking, bold, half-bright, reverse video, struck-through, and underline.

Font

You can create workstation fonts using one of the font design applications provided. For character-map workstations, use the Font Designer; for bit-map workstations, use the Raster Font and Icon Designer.

The font contains pixel information for all 256 characters. Character-map workstations also support half-pixel shift in any pixel row of a character. This allows the Font Designer to maximize resolution.

Cursor

On a character-map workstation, the *standard cursor* is a blinking underline. On workstations with enhanced video capability, a block cursor can also be selected through a system configuration file option. Bit-map workstations have a software-loadable cursor. The cursor bit array is superimposed in the character.

Video Refresh

On character-map workstations, the video RAM is contained within the processor module and is accessible to the processor at a fixed location in the processor's address space. The location of the character map cannot be changed. To switch screens, it is necessary to copy the contents of the character map.

Writing Portable Video Programs

Different workstation models have different numbers of lines on the screen. Therefore, care must be taken to write code that can run on a screen with a variable number of lines. This type of code can be written as follows.

During initialization, include a call to `QueryVidHdw` or `QueryVideo`. (For details on these operations, see the *CTOS Procedural Interface Reference Manual*.) The memory address of a block of video information is returned. At offset 1 in this block is a 1 byte field called *nLinesMax*. This field contains the number of lines on the screen. Lines are numbered starting with line 0.

When writing calls to operations that require row and column coordinates (such as `PutFrameChars` or `PutFrameAttrs`), the row coordinate should be used as a variable rather than as a constant.

For example, to write a message on the line of the screen that is 2 lines from the bottom, the row coordinate used is *nLinesMax-3*.

Video Data Structures

The video control block (VCB) contains all information known to the operating system about the video display, including the location, height, and width of each frame, and the coordinates at which the next character is to be stored in the frame by SAM. You can obtain the memory address of the VCB by calling the GetPStructure operation with a *structCode* value of 2. (See the description of GetPStructure in “Operations” in the *CTOS Procedural Interface Reference Manual*. For the format of the VCB, see “Video Control Block” in “System Structures,” in that same manual.)

The VCB contains an array of frame descriptors. A frame descriptor is a component of the VCB and contains all information known about one of the frames. The number of frame descriptors in the VCB is specified at system build. (For the content of a frame descriptor, see “Frame Descriptor” in “System Structures” in the *CTOS Procedural Interface Reference Manual*.)

Programming Using Color

If you want to use color in your programs or if you want to program the graphics control registers, you must use the ProgramColorMapper operation. (For details and examples of how this is done, see the *CTOS Programming Guide*.)

Video Operations

The video operations described below are categorized by software function. The VAM operations are arranged alphabetically. The remaining operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

VAM Operations

FillFrame

Is similar to `PutFrameAttrs` except `FillFrame` fills a specified frame with the same character (rather than the same character attribute) for a range of character positions within the frame.

FillFrameRectangle

Fills a specified rectangle within a specified frame with characters and corresponding attributes.

FillInRectangle

This operation provides the same functionality as `FillFrameRectangle` but can be called by applications running on real mode operating systems.

FrameBackspace

Searches backwards from the location specified by the frame coordinates in the specified frame to find the first non-null character.

MoveFrameRectangle

Moves an arbitrary rectangle of characters and corresponding attributes within a frame of the character map to another position in the map.

PosFrameCursor

Establishes a visible cursor within the specified frame at the specified coordinates.

PutCharsAndAttrs

Combines the PutFrameChars and PutFrameAttrs functions so that a sequence, such as a line of characters and attributes, can be written in a single call. This operation is similar to PutFrameCharsAndAttrs except that it is a standard operating system library procedure that works on all operating system versions.

PutFrameAttrs

Establishes the same character attribute for a range of character within a specified frame.

PutFrameChars

Overwrites the specified character positions in the specified frame with the specified text string.

PutFrameCharsAndAttrs

Combines the PutFrameChars and PutFrameAttrs functions so that a sequence of characters and attributes can be written in a single call.

QueryBounds

Returns the size (number of columns and lines) for the specified frame.

QueryCharsAndAttrs

Returns a character string and its associated attributes from the character map at the specified coordinates.

QueryCursor

Returns the cursor position for the specified frame. This operation is similar to QueryFrameCursor except that it is a standard operating system library procedure that works on all operating system versions.

QueryFrameBounds

Returns the size in number of columns and lines for the specified frame.

QueryFrameChar

Returns a single character located in the character map at the specified coordinates of the specified frame.

QueryFrameCharsAndAttrs

Returns a character string and its associated attributes from the character map at the specified coordinates.

QueryFrameCursor

Returns the cursor position for the specified frame.

ResetFrame

Restores the frame to its initial state; that is, all character positions are blanked and all character attributes are reset.

ScrollFrame

Scrolls the specified portion of the specified frame up or down by the specified number of lines.

Video Requests

QueryFrameAttrs

Returns an attribute string in the virtual character map beginning at the specified coordinates of the specified frame.

QueryFrameString

Returns a character string in the virtual character map beginning at the specified coordinates of the specified frame.

VDM Operations

QueryVidHdw

Places information describing the level of video capability of the workstation in the specified memory area. QueryVidHdw fills in only certain fields in the specified memory area according to the operating system version.

QueryVideo

Performs the same function as *QueryVidHdw* except it fills in all fields in the specified memory area. This operation is a standard operating system library procedure that works on all operating system versions.

LoadFontRam

Reads the character font from the specified open file to the specified memory area and then transfers the font to the font RAM.

ResetVideo

Suspends video refresh, resets all screen attributes, and changes the values stored in the VCB to reflect the specified parameters. All frames and VCB fields (except the flag *fReverseVideo*) are initialized, and the flag *fExecScreen*, in the Application System Control Block is set to FALSE. Additionally, *ResetVideo* may cause a new font to be loaded on character-map workstations with EV.

ResetVideoGraphics

Functions in the same way as *ResetVideo* but allows the caller to set the graphics bit-map resolution, the number of planes for color, and the background color mode on workstations with VGA capability.

InitVidFrame

Defines the screen coordinates and dimensions of one of the frames.

SetScreenVidAttr

Sets/resets a specified screen attribute.

InitCharMap

Initializes the character map.

SetVideoTimeout

Causes the screen refresh to turn off after a specified time has elapsed during which no keyboard activity has occurred.

Color Programming Operations

ProgramColorMapper

Sets and queries the palette and/or the control structure.

SetAlphaColorDefault

Sets up a default alpha palette and control structure.

LoadColorStyleRam

Specifies 8 bytes that are passed to the color graphics style RAM. These attribute settings display different combinations of color, reverse video, and underlining.

SetStyleRam

Sets a flag that indicates which of the following style RAMs is to be used: the graphics style RAM or the standard alphanumeric style RAM.

SetStyleRamEntry

Modifies a single 1 byte entry in the graphics style RAM.

LoadBackgroundPalette

Specifies an array of bytes to be used as a color palette for background colors.

Direct Access to Video Data Structures

It is possible, although not recommended, to access the video data structures directly at an interface level below VAM and VDM. Although programming at this lower level can be more efficient than using VAM or SAM, your application will not be compatible among the several workstation models. Specifically, your application will not work on a bit-map workstation.

To write directly to the character map, use the `rgpVidMemLine` structure (GetPStructure, *structCode* 7).

The following operations provide direct access to the video data structures:

LockVideo

Locks the video structures used by the operating system.

UnlockVideo

Is used after calling `LockVideo`, to remove a lock on the video structures used by the operating system.

LockVideoForModify

Modifies the video structures used by the operating system.

UnlockVideoForModify

Is used after `LockVideoForModify` is called to remove a lock on the video structures used by the operating system.

Section 11

Keyboard and I-Bus Management

What is Keyboard Management?

Keyboard management allows you to write programs that can read the keyboard to obtain detailed information about each keyboard event and the current keyboard state.

Keyboard management provides many options. We suggest that you not try to absorb all of them in one reading. You should start by reading “Keyboard Modes” and “Reading the Keyboard.” These sections describe ways your application can read keyboard input. You should also be aware that there are two keyboard hardware protocols, one for CTOS-style keyboards (for example K1, K2, and K5) and another for PC-Style keyboards. These differences are noted in “Keyboard Hardware Protocols” and are mentioned in “Keyboard Management Overview.” To consider ways you may be able to use some of the keyboard features in future applications, you also may be interested in reading “Keyboard Management Features.”

Later, you can investigate the sections describing keyboard translation and emulation data blocks. The keyboard data blocks really are the driving force behind keyboard management. They ultimately define keyboard events. The keyboard process merely provides the facility to handle keyboard events based on the contents of the data blocks.

Unless you are writing very specialized applications, chances are you will not need to know about keyboard data blocks in great detail. The operating system loads a default set (one for translation and one for emulation) at system initialization. However, the more you know about them, the more tools you will have to work with in the future. For example, you can use data blocks to customize the way applications work locally or on a global basis, to meet keyboard hardware and nationalization requirements. You can perform customization either dynamically or by using the Keyboard Customizing utility provided with the Standard Software Development Utilities. Perhaps the most

requirements of your application without changing the operating system code.

Keyboard Terminology

Terms relating to keyboard management are described in Table 11-1. You may need to refer to this table from time to time as you read about different keyboard topics.

Table 11-1. Keyboard Terms

Term	Definition
Allowed state	For a nonchord key, the chords that influence the character code generated. Some chords can be pressed but do not have an effect on the resulting character displayed. (SHIFT+LOCK , for example, affects only the alpha keys.) For a chord key, the allowed state determines whether a chord on a physical keyboard is allowed on a target keyboard. (See <i>Emulating</i> .)
Application profile	The keyboard an application expects is being used. (See also <i>System profile</i> .)
Character code	The translation of the unencoded value and the chord state of a key. A character code can be 8, 16, or 32 bits long.
Character mode	Mode in which an application reads keyboard events and is returned the displayable character code.
Chord	A key that, when pressed, can influence the displayable character code generated by subsequent keys pressed. Examples are SHIFT , LOCK , CODE , and ALT . Chords do not generate character codes in character mode.
Chord state	A word that is a bit map of the chord keys currently pressed.
Decoding value(s)	An ordered set of unencoded values that will produce a character code.

contin

Table 11-1. Keyboard Terms (cont.)

Term	Definition
Diacritics	Keys working in pairs. When pressed in the proper order they produce a diacritical result such as a German a with an umlaut. Pressing the first key enables diacritical mode (nothing is returned); pressing the second returns the result.
Emulating	Mapping the chord state and unencoded values on the source keyboard hardware to the chord state and unencoded values necessary to produce the same character on the target keyboard.
Extended character set	A character set containing more than 256 characters. Characters need to be defined by more than one byte.
I-Bus style	Keyboards using the I-Bus protocol. Most CTOS keyboards are in this class and include the K1 through K5 keyboard and the SuperGen I-Bus keyboard (SG101-K).
I-Key	A keyboard value used to index various keyboard data block tables to locate information about the key. The I-key may be a key post value generated by the hardware, or it may be an emulated value.
Keyboard code	For I-Bus style keyboards, is an 8-bit value describing the keyboard event. When the high-order bit is 0, the 7 low-order bits indicate the hardware key post downstroke. When the high-order bit is set, the value indicates the key post upstroke. A different protocol is used for keyboards not using the I-Bus. (See "Keyboard Hardware Protocols" for details.)
Keyboard data block	The smallest complete set of data supporting translation or emulation. Although keyboard data blocks are alternately called <i>translation</i> or <i>emulation tables</i> , they contain a variety of structural components (including translation or emulation subtables) that support the translation or emulation process. The file <i>NlsKbd.sys</i> is comprised of keyboard data blocks.
Keyboard event	A key downstroke or upstroke.

continued

Table 11-1. Keyboard Terms (cont.)

Term	Definition
Multibyte character	A character defined by more than one byte.
Multibyte strings	A sequence of keys generated when a single key is pressed.
PC-style	A keyboard (used with the SuperGen Series 5000, for example) that does not use I-Bus protocol (SG102-K). Other documentation may refer to this keyboard as the <i>PS/2-style</i> keyboard.
Raw unencoded	The unencoded value of key returned from the attached hardware.
Standard character set	The single byte character codes. See Appendix B in the <i>CTOS Procedural Interface Reference Manual</i> .
System profile	The keyboard emulated on an operating system. (Applications running on the operating system assume this keyboard is the one being used.)
Toggle	A characteristic of some chord keys (for example LOCK) where the key maintains an influence on other keys after it has been pressed and released. To return to its original state (no influence on other keys), the toggle must be pressed a second time. By default, chords that do not toggle (for example SHIFT) maintain their influence only while pressed.
Translating	Generating the character code from an unencoded value and the chord state.
Type-ahead buffer	A partition structure that stores unencoded keyboard values to be returned directly or after translation to an application. The type-ahead buffer also stores other input event types such as mouse events.
Unencoded value	A keyboard code. Unencoded values are not translated values but may be the result of emulating the hardware keys of another keyboard. A raw unencoded value is not translated.
Unencoded mode	Keyboard mode in which an application reads the keyboard and gets back an unencoded value for the keyboard event.

Keyboard Management Features

Keyboard management provides the following features to users.

Generic Design

Keyboard management is designed to accomodate CTOS and BTOS keyboard requirements and is intended to support future keyboards with requirements yet to be known.

Reduces Future Operating System Changes

The removal of decisions from the keyboard process and placement of this information in external, user definable data blocks is a design element to reduce operating system source code changes. Existing keyboard data blocks can be changed and instances of existing data blocks with modifications can be added without altering the keyboard logic.

Easy to Customize

The system keyboard file, *NlsKbd.sys*, is created by linking together object modules, each of which contain keyboard data blocks for a specific keyboard. You can easily customize this file by changing which keyboard data blocks are linked. In addition, you can use the Keyboard Customizing utility packaged with the Standard Software Development Utilities to create object modules containing keyboard data blocks for a given keyboard that have been altered to suit the requirements of your application.

Supports Nationalization and Extended Character Sets

Each of the characters in the standard character set is uniquely defined using a single byte. The standard single-byte character definition, however, is not adequate to define character sets in excess of 256 characters. (The Japanese character set, for example, contains 6802 characters.) To broaden nationalization capabilities, keyboard management supports multibyte characters in extended character sets. Currently characters can be defined using 1, 2, or 4 bytes.

Supports Multiple Users

Applications at cluster workstations can share the same system keyboard file at the server and run with different local physical keyboards. Applications on a given workstation can use different keyboard data blocks managed by user number.

Simplifies Writing Applications

An application does not need to read the keyboard in unencoded mode to obtain information about unencoded key values. Applications written in character mode have the option of getting unencoded information with the ReadKbdInfo operation. The only time an application must be written in unencoded mode is when there is a need for the application to oversee all keyboard events such as setting LEDs and managing typematic keys.

Provides Services For Persons With Disabilities

The keyboard process provides features to help persons with disabilities. As examples, chord keys and ACTION are allowed to remain in effect after being released until the user presses and releases a character key. Character repeating can be slowed or stopped. Strings, diacritics, and chords as well as single characters can be repeated. The state of a toggle chord (on or off) can be determined by its audio tone.

Provides Backwards Compatibility With Old Programs

Existing applications can run without modification. Applications, however, will benefit from using the ReadKbdInfo operation instead of older keyboard requests for reading keyboard data.

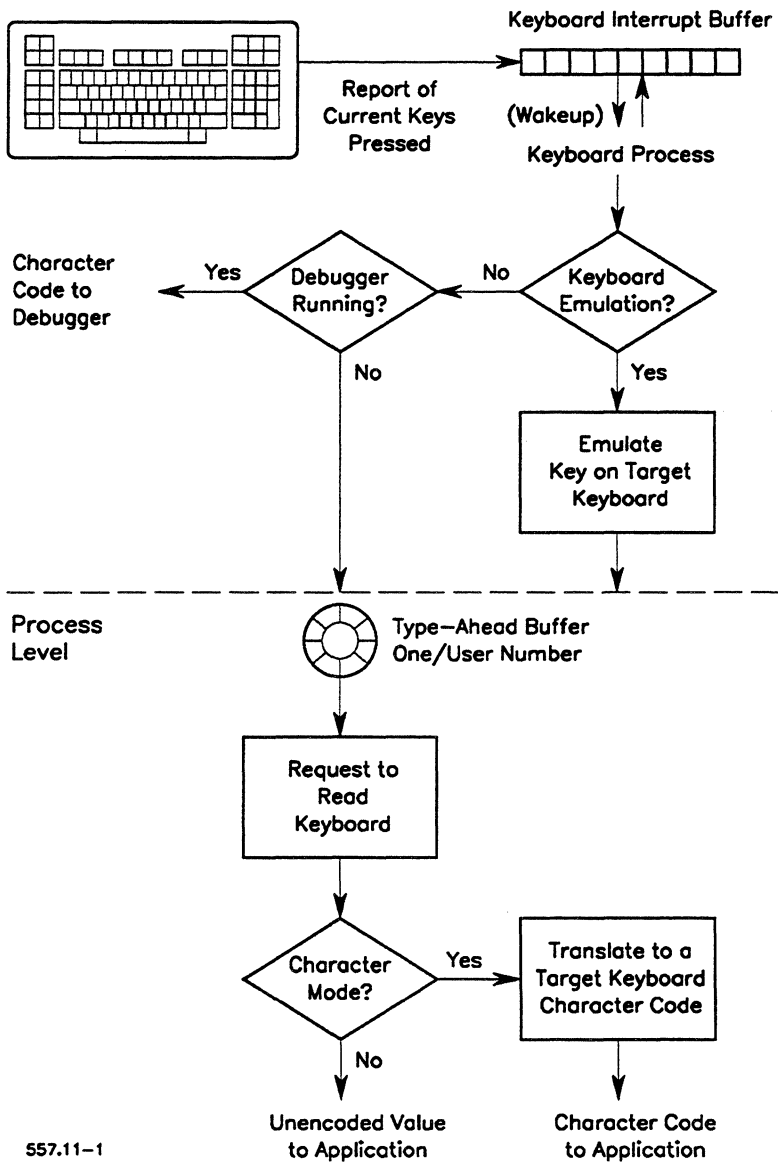
Keyboard Management Overview

Figure 11-1 is an overview of what happens from the time a key is pressed or released until the keystroke is placed in application memory. The manner in which keyboard events get placed in the internal buffer is different for keyboard hardware that does not use the I-Bus. (See “Keyboard Hardware Protocols” for details.) However, this detail is inconsequential to the main focus here: how keyboard processing works to get keyboard data to an application.

Note: *The overview is a simplification. For example, an application can obtain all the details of a keyboard event regardless of the mode in which it reads the keyboard. Keyboard modes, however, are discussed in other sections. (See “Keyboard Modes” and “Reading the Keyboard.”)*

Keyboard management provides the facility for preprocessing and postprocessing of keyboard events.

Figure 11-1. Keyboard Management Overview



557.11-1

Preprocessing

Preprocessing gets key events to the appropriate type-ahead buffer. (See Figure 11-1.) Preprocessing activities are shown in the region above the dashed line.

Hardware actions leading up to keyboard processing start with the user pressing or releasing a key. This triggers a hardware interrupt. The keyboard microprocessor transmits the event to the keyboard interrupt service routine as a sequence of bytes indicating all keys currently pressed.

On I-Bus style keyboards, the interrupt service routine maintains a log of all keys pressed. It compares this data to the current hardware report on keys pressed. The difference is the key post value of the affected key. Having identified the key, the interrupt routine sets the top bit of the keyboard code if the event was an upstroke. Then it places the keyboard code in the keyboard interrupt buffer and wakes up the keyboard process.

When notified to wake up, the keyboard process determines the user number of the current keyboard owner and ensures that the appropriate keyboard data blocks are used.

If emulation data is involved, the keyboard processes uses the appropriate emulation data block to map the keyboard code from the source keyboard to the keyboard code on the target keyboard that would ultimately translate to the same displayable character. As a result of emulation, the *mapped unencoded* value is placed in the type-ahead buffer of the appropriate user number. If there is no emulation, the *raw unencoded* value (key post value directly from the attached keyboard) is placed in the type-ahead buffer.

Placing the keyboard code in the type-ahead buffer completes the preprocessing activities.

See Figure 11-1. If the Debugger is loaded, the keyboard process directs the keyboard events to the Debugger instead of to the application. Because the Debugger reads keyboard events in character mode, the keyboard process must first translate the keyboard code to a character code before passing it to the Debugger.

Postprocessing

Postprocessing delivers key events to the application in a comprehensible form.

See Figure 11-1 (above). Postprocessing is shown in the region below the dashed line. When an application reads the keyboard in unencoded mode, it receives the keyboard code directly from the type-ahead buffer.

If the application reads the keyboard in character mode, the keyboard processes uses the appropriate translation data block to translate the keyboard code a character code, which it passes to the application.

Keyboard management has worked much as it is described here in previous operating system versions. Translation and emulation, however, are not an integral part of the keyboard code. All descriptions of the keyboard state and of how a key ultimately is displayed are contained in the keyboard translation and emulation data blocks. Preprocessing and postprocessing simply involve the keyboard process consulting these keyboard data blocks and using the rules contained therein to return the appropriate key values to the application.

Keyboard Modes

An application program can set the keyboard mode using the `SetKbdUnencodedMode` operation. Although `SetKbdUnencodedMode` provides several keyboard mode variations (see “Other Keyboard Modes”), the keyboard is read in two basic modes: unencoded mode and character mode (the default). These are described below.

Unencoded Mode

When the user presses or releases a key, the event generates a keyboard code for the downstroke and for the upstroke. The interpretation of the downstroke and upstroke values depends on the hardware protocol used. (See “Keyboard Hardware Protocols.”) Typically, (with the I-Bus hardware protocol) the 7 low-order bits of the 8-bit keyboard code identify the key post. The top bit is set to indicate the upstroke. (See Appendix C in the *CTOS Procedural Interface Reference Manual*. The appendix lists the 7 bit downstroke values generated for each key on a K1 keyboard.)

The downstroke and upstroke of a key are returned in separate read requests to an application reading the keyboard in unencoded mode.

As an example of what the application sees in unencoded mode, let’s suppose the user wishes to display an uppercase **A** from a K1 keyboard. The user performs the event sequence shown below, and the program receives the corresponding keyboard codes:

Keyboard Event	Keyboard Code
1. Press LEFT SHIFT .	48h
2. Press A .	61h
3. Release A .	E1h
4. Release LEFT SHIFT .	C8h

Even though there are two shift keys on a K1 keyboard, each hardware key post has a unique value. **LEFT SHIFT** and **RIGHT SHIFT**, for example, are identified by different keyboard codes.

Character mode

In *character mode* (the default keyboard mode), the program receives a character code when a key other than a chord key or **ACTION** is pressed. (Chord keys include **SHIFT**, **CODE**, **LOCK**, **NUM LOCK**, and **ALT**.) Pressing a chord key does not generate a character code, but influences the displayable value generated by other keys pressed simultaneously. **ACTION** has a special, system-wide meaning. (For details, see “Action Key.”)

For the four-event key sequence needed to display the uppercase A (see “Unencoded Mode”), a program in *character mode* would receive one character code, the code for uppercase A.

Character mode provides the program with the same kind of information as a traditional n-key rollover encoded keyboard but with even greater flexibility. As the keyboard process converts sequences of unencoded keyboard values to character codes, it accesses a variety of other information about the keystroke and the keyboard state. If an application uses the `ReadKbdInfo` operation to read from the keyboard, it can run in character mode and still obtain the unencoded values needed to generate the character code. (See “Reading the Keyboard.”)

Other Keyboard Modes

In addition to unencoded mode and character mode, the `SetKbdUnencoded` operation allows an application to read the keyboard in the following variations of these modes:

- Mapped unencoded
- Raw unencoded
- Unencoded plus
- Character plus

In *mapped unencoded* mode, the unencoded values returned from a keyboard are mapped to the corresponding unencoded keyboard codes for the target keyboard. These mapped unencoded values are placed in the application type-ahead buffer. The application never sees the values actually generated by the attached keyboard. (See Table C-1 in the *CTOS Procedural Interface Reference Manual*.) Typically, applications reading the keyboard in unencoded mode receive mapped unencoded values.

In *raw unencoded* mode, the keyboard returns the unencoded values generated by the attached hardware: the values do not undergo emulation.

In *unencoded plus* mode, the application receives both the raw unencoded and the mapped unencoded key values. *Character plus* mode returns both the raw unencoded values and the character code. In either of these modes, two reads must be performed to retrieve all the data. The raw values only are returned with first read.

Comparing the Keyboard Modes

In most cases an application can run in character mode and (using the `ReadKbdInfo` operation) obtain all the information it needs about the keyboard state, which keys are pressed, when an event is an upstroke, and so forth. For this reason, unless there are unusual circumstances, applications do not need to use the other keyboard modes. Like character mode applications, applications running in any of the unencoded keyboard modes can have unencoded values translated to character codes. However, such applications are responsible for managing their own LEDs and character repeating. (See “Reading the Keyboard.”)

Applications running in raw modes must be hardware aware. They need to call the `GetKeyboardId` operation to obtain the identification number of the current keyboard. Once obtained, they need to know the keyboard layout. Such applications are totally responsible for correctly interpreting the raw values the keyboard generates. Raw key values differ, depending on the keyboard hardware protocol employed.

Keyboard Hardware Protocols

The operating system supports two hardware input device protocols: the Input-Bus (I-Bus) protocol and the PC-Style keyboard protocol.

The I-Bus protocol is used for I-Bus style keyboards. It identifies each hardware key post with an 8-bit value. A high-order bit of 0 means the key post is down (pressed by the user); a high-order bit of 1 means the key post upstroke.

The PC-style keyboard protocol is different. It employs 8-bit values up to value EFh for the downstroke. The upstroke is the value F0h *xx*, where *xx* is the downstroke value. Applications reading the keyboard in raw unencoded and unencoded plus mode are responsible for correctly interpreting the protocol used.

Reading the Keyboard

It is recommended that an application use the `ReadKbdInfo` operation to read keyboard data. To receive keyboard data and other input events (such as mouse data when Mouse Services is installed), the application should use `ReadInputEvent`. Applications that use these interfaces for keyboard input can read multibyte characters in extended character sets, facilitating portation of these programs to other nationalized operating systems. (For details, see the section entitled “Native Language Support.”) The operations allow a program reading the keyboard in any keyboard mode to receive input about each keyboard event as well as the translated character.

Note: *To read submit file input, an application must use `ReadKbdInfo` rather than `ReadInputEvent`. When using `ReadKbdInfo`, Mouse input is not available to such applications. (See the CTOS Programming Guide for details on Mouse Services.)*

Using `ReadKbdInfo`

Using `ReadKbdInfo` an application can set selected bits in the *mode* parameter to obtain the following information:

- Setting bit 0 (block/ no block) causes the keyboard process to wait until there is keyboard data to return.
- Setting bit 1 (peek/read) leaves the keyboard data in the type-ahead buffer.
- Setting bit 4 (no filter/filter) directs the keyboard process to return the unencoded value, whether or not a character code is generated. In character mode, an application that does not set this bit just has information returned if the event produces a character code. Whether or not this bit is set, unencoded applications receive unencoded values returned by this operation.

- Setting bit 5 directs keyboard input to the keyboard process exchange. When bit 5 is 0 and the application is reading the keyboard in character mode, input is filtered by the system input process. (For details, see “System Input Process.”)

In addition to returning the unencoded value or the character code, `ReadKbdInfo` returns other detailed information about the keyboard event and the keyboard state. (See below.)

Information Returned to the Requesting Application

Keystroke information returned by `ReadKbdInfo` includes the following:

- The chord state (bit map of chord keys currently pressed, if any)
- The unencoded key value (can be an emulated value)
- The character size (in bytes) and character code (if one is generated)
- Additional detailed keystroke information passed back to the application in two status words (see below)

Status Word Information

The first status word *wStatus* contains information available to the application writer through the `ReadKbdInfo` interface. Details it returns are described in the text that follows. The second status word *wStatus1*, however, is reserved for programmers writing their own drivers. It is not described here.

For applications programmers, *wStatus* defines detailed information about each keystroke, such as if

- It is the result of a diacritic translation
- It follows a diacritic key
- It begins a diacritic key pair
- It is the first character in a character string
- It generates a character code
- The keystroke it is a chord
- It is an upstroke

- The character generated is a result of holding down a typematic key
- The character generated has typematic capability
- The character generated follows a diacritic start character but does not match one of the possible diacritic end characters
- The keyboard code is the raw unencoded value

Other information that can be returned in *wStatus* includes the state of the user bit and the command bit. Both of these bits are for the convenience of the application. The user bit is reserved for applications that want to define a key as having special significance for all chord states; the command bit allows the application to define the combination of this key with the current chord as having special significance.

To set the user bit or the command bit for a key, you can edit data block text files destined for processing using the **Create Keyboard Data Block** command. (For details, see the *CTOS Executive Reference Manual*.) The user bit and command bit can also be set dynamically by an application familiar with keyboard data block structure. (See “Modifying Data Blocks” for details.)

The command bit can increase the efficiency of applications currently using internal tables to define application-specific commands. With every keystroke, for example, an application may scan an internal commands table to see if the key is present. Using the command bit to define command keys, the application would only need to scan the table when it knew it had a command key.

Type-Ahead Buffer

The *type-ahead buffer* stores unencoded values not yet returned to an application. It is part of the User Structure of a partition and is created when the partition (user number) is created.

If a multibyte string is defined for the unencoded key value, the keyboard process expands the string before placing it in the type-ahead buffer. Multibyte strings can be embedded in translation data blocks and emulation data blocks.

String expansion is actually an extra step to the preprocessing and postprocessing illustrated in Figure 11-1. (See “Keyboard Management Overview.”) Before placing unencoded key values in the type-ahead buffer, keyboard management looks ahead to the translation data block to determine if any of the values should be further expanded. If a multibyte string is generated, the unencoded values resulting from string expansion are placed in the type-ahead buffer.

If too many unencoded values are placed in the type-ahead buffer before processing, the excess values are discarded. When a program reads beyond the values buffered successfully, it receives status code 602 (“No character available”). By default, the size of the type-ahead buffer is 128 bytes but the size can be changed at system build. If an application terminates with an abnormal (nonzero) status code, the contents of the type-ahead buffer are discarded by the Chain and ErrorExit operations. (See “At Program Termination.”) The contents of the type-ahead buffer are also discarded when the application posts its own data blocks (see “Posting Data Blocks”) and when the attached keyboard hardware changes.

Writing to the Type-Ahead Buffer

Using the WriteKbdBuffer operation, an application can write keyboard events into the type-ahead buffer of a specified user number.

WriteKbdBuffer typically is used by a partition managing program.

Context Manager, for example, uses WriteKbdBuffer to accomplish its Cut And Paste function. It decodes display characters it takes from the transmitting context. (See “Decoding.”) Then it calls WriteKbdBuffer to write the unencoded keystrokes directly into the type-ahead buffer of the receiving context. WriteKbdBuffer gives the context managing program the option of writing unencoded values until the buffer is full or of having status code 619 (“No buffer space”) returned if the buffer is not large enough to hold all the data. WriteKbdBuffer also disables or enables direct keyboard or mouse input. In addition it can support remote debugging.

Keyboard Data Block Organization

The system keyboard file *NlsKbd.sys* contains a linked set of *keyboard data block* objects. A keyboard data block is the smallest complete set of data defining keys for a keyboard type, such as a K1 or a K5 keyboard. It contains all the structures necessary to support either emulation or translation. Emulation data blocks map keys from a source keyboard to a target keyboard. Translation data blocks translate keyboard codes to displayable character codes. The system keyboard file can contain up to 64K bytes of customized keyboard data blocks.

Note: *The compilable objects in NlsKbd.sys are alternately known as translation or emulation tables and the components, as subtables. The text uses somewhat different terminology. The compilable objects are referred to as translation or emulation data blocks and the components for translating or emulating, as tables. All other components in the data block that help define the key (structures, bit masks, and so forth) are called supporting tables.*

The components of a keyboard data block and their layout are similar for translation and emulation. Figure 11-2 shows the general layout.

The data block header contains several fields including the data block signature, the keyboard ID, and offsets to other tables in the data block. The translation data block header shown in Figure 11-3 (see “Translation Data Block Organization”) illustrates some of these fields in more detail.

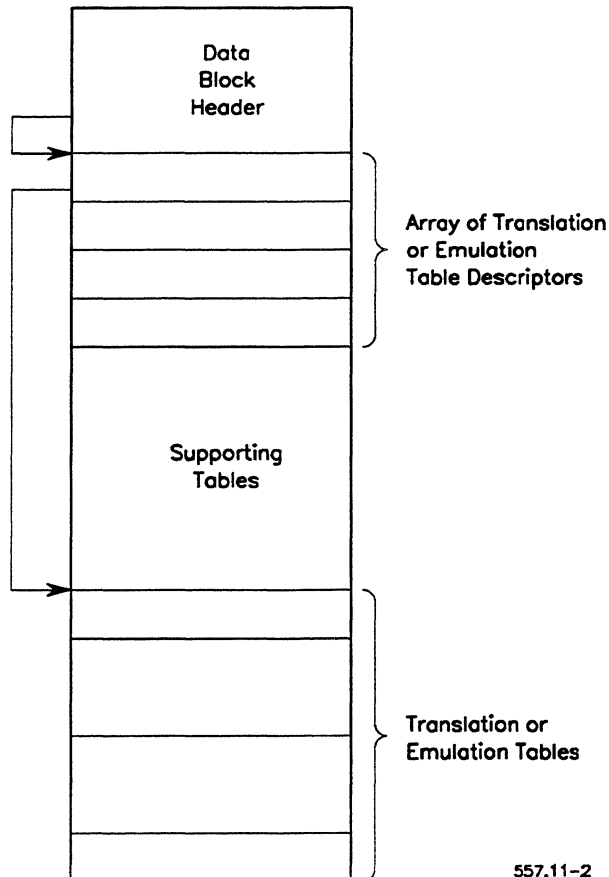
Note: *Figure 11-2 shows the relative location of the data block components to each other in memory. For details on the formats of translation and emulation data blocks and on the supporting components, see “System Structures” in the CTOS Procedural Interface Reference Manual.*

Offsets in a keyboard data block header point to other components in the data block. Figure 11-2 shows one offset pointing to the beginning of an array of table descriptors. In an emulation data block, each table descriptor is a word value containing the offset from the start of the data block to an *emulation table*. In a translation data block, each descriptor is two words: the first word contains an offset to a *translation table*; the second contains the character size defined in that table.

Figure 11-2 shows four table descriptors. Each descriptor offset points to one of the four (translation or emulation) tables in the lower region of the data block. There can be more or fewer tables, depending on the keyboard being used and customizations.

The translation or emulation data block header also contains offsets to *supporting tables*. (For details on supporting tables, see “Translation Table Data Block Organization” and “Comparing Data Blocks.” See “Modifying Data Blocks,” for the format of the *NlsKbd.sys* table of contents at the beginning of the system keyboard file.)

Figure 11-2. General Data Block Layout



557.11-2

Finding a Key Definition

With each keyboard event, the keyboard process uses the key post value (commonly called the *I-Key*) as an index into various supporting tables to look up information leading to the key definition.

By following some rules, the keyboard process arrives at the appropriate visual representation of the key. (See “Translating” for details.) In most cases there is a one-to-one correspondence between the keyboard event and the unencoded values generated. Some exceptions are keys involved in diacritics, keys generating multibyte strings, and keys absorbed as a result of emulation.

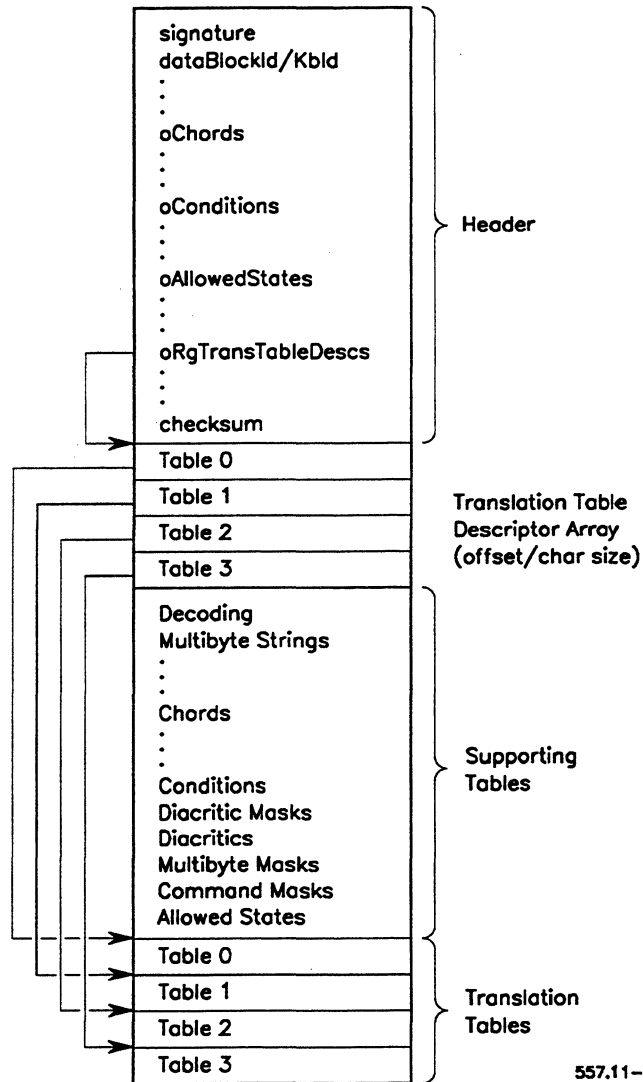
Translation Data Block Organization

Translation maps keyboard codes to character codes. Figure 11-3 shows the layout of a typical translation data block for the K1 keyboard.

One field in the header, *oRgTransTableDescs*, points to the start of the translation table descriptor array. Each array element (word) contains the offset from the start of the data block to a translation table and the character size (one, two, or four bytes) defined in the table. Other header fields are offsets to supporting tables. For example, *oChords* and *oConditions* are offsets to the Chords and Conditions supporting tables, respectively. (See “Supporting Tables,” for details.)

The Keyboard Customizing utility packaged provides a text file version of the K1 keyboard data block. Using a text editor, you can open this text file and compare its contents to the layout shown in Figure 11-3. The file contains comments explaining how to interpret the contents of the translation tables and the supporting tables. (For additional information on each data block structure, see “System Structures” in the *CTOS Procedural Interface Reference Manual*.)

Figure 11-3. Translation Data Block Format



Translating

***Note:** Although emulation tables can have up to 240 values (0 to EFh), there are a maximum of 128 entries (0 to 7Fh) in any one translation table. This is because the operating system and existing applications assume the high-order bit of an 8-bit keyboard code, when set, means the upstroke.*

A translation table contains an entry for each key post value, up to a maximum of 128 entries. The entry provides the corresponding ASCII character code returned to an application in character mode. Guided by a table selection process, the keyboard process selects the appropriate translation table based on the key post value and the current chord state (whether **SHIFT** and/or **CODE** keys are also pressed).

In the U.S. default version of the system keyboard file, there are four, 128-entry K1 keyboard translation tables. Each table defines a one-byte character generated for a K1 key post under one of the following conditions:

- No chord keys are pressed (Table 0).
- **LEFT SHIFT** or **RIGHT SHIFT** or **SHIFT+LOCK** is pressed (Table 1).
- **LEFT CODE** or **RIGHT CODE** is pressed (Table 2).
- At least one **SHIFT** key and one **CODE** key are pressed (Table 3).

As an example, if **A** is pressed with **RIGHT SHIFT**, this condition selects Table 1. (See “Selecting the Translation Table to Use.”)

Supporting Tables

Figure 11-3 shows tables just beneath the translation table descriptor array. These are tables supporting translation. Offsets to these tables are contained in the data block header. The *supporting tables* define key functions and other information about the key, such as whether the key generates a multibyte string or if it is the first key of a diacritic key pair. The tables include multibyte strings, special keys, chords, repeat attributes, allowed states, conditions (selecting a translation table), diacritics, and command masks.

The supporting tables for emulation are similar in content and number to those for translation. (See “Comparing Data Blocks” for a comparative list of all the supporting tables.) The offsets from the beginning of the data block to each of these tables is contained in the data block header.

In the translation data block, the keyboard process uses the supporting tables to perform the following functions:

- To determine which of the translation tables it will ultimately use to translate keyboard codes to character codes
- To obtain other details about the current key pressed and the keyboard state to pass back to requesting applications through the `ReadKbdInfo` (or `ReadInputEvent`) interface

Selecting the Translation Table to Use

To locate the correct translation table, the keyboard process consults the following three supporting tables:

- Chords
- Allowed states
- Conditions

The chords table defines the state (on or off) of chord keys. The partition keyboard information structure maintains this chord state information in the bits of the state word. (See “System Structures” in the *CTOS Procedural Interface Reference Manual*, for the format of this structure.) The chords table also indicates whether chords have LEDs and if they are defined to function as toggles.

The allowed states table determines which chords influence translation of the key. (For example, `LEFT SHIFT`, `RIGHT SHIFT`, `LEFT CODE`, `RIGHT CODE`, and `LOCK` affect alphabetic K1 keyboard keys but `LOCK` has no affect on the arrow keys.)

The conditions table describes the table selection process in the data block.

***Note:** If a key is a chord, its allowed state in the translation data block determines whether the chord will be placed in the application type-ahead buffer. If the bit representing the chord is not included in the allowed state, the chord key will be used for determining emulation of keys pressed in combination with it but the actual chord key value is not passed through to the application.*

Let's assume a user has pressed a key on the K1 keyboard. Starting with the key post value and the chord state of the key, the keyboard process uses the following procedure to select the table to use to translate the key:

1. It uses the chords table to determine if the key is a chord and which chords are currently pressed.
2. It uses the key post value (*I-Key*) as an index into the allowed states table to determine which chords influence translation of the key. By comparing the chord state (from step 1) with the allowed states, it determines the effective chord state.
3. It consults the conditions table to compare the effective chord state with different sets of conditions. The conditions set that matches the effective chord state specifies the appropriate translation table.

As an example, suppose **LEFT SHIFT** and **RIGHT SHIFT** are the only effective chords. This means either shift key will influence the resulting key translation (but the state of any other chords or chord combinations such as **RIGHT CODE**, **SHIFT+LOCK**, or **LEFT CODE** will not). If the chord state indicates that **LEFT SHIFT** is pressed, the Conditions table directs the keyboard process to use the translation table for the condition that **LEFT SHIFT** or **RIGHT SHIFT** or **SHIFT+LOCK** is pressed.

Obtaining Other Keyboard Details

The keyboard process also can obtain other details about the current keystroke from the keyboard supporting tables. For example, the diacritic masks, command masks, and multibyte masks tables identify which translation tables (if any) define the key as a diacritic, a command key, or a key generating a multibyte string.

***Note:** The keyboard data blocks released as the contents of the default U.S. system keyboard table, `NlsKbd.sys`, are engineered to duplicate an environment that is backwards compatible with older applications. Nationalized keyboard data blocks, however, are customized to meet the requirements of nationalized applications. For details on nationalization, see Chapter 42, “Native Language Support.”*

Contents of the Default K1 Translation Data Block

The default K1 keyboard translation data block in `NlsKbd.sys` is designed to reflect the 8 bit superset of the ASCII printable characters in the standard character set. (See Appendix B in the *CTOS Procedural Interface Reference Manual* for the contents of the standard character set.)

The values of the K1 keys are preserved in the U.S. default K1 translation data block for backwards compatibility with older programs reading the keyboard. For example, the translated value of **LEFT ARROW** is Eh in K1 translation table 0 (no chord keys pressed). The value of **CODE+LEFT ARROW** is 8Eh in translation table 2 (either code key is pressed). (See “Translating” for descriptions of the default K1 keyboard files.) The values Eh and 8Eh are also the respective character codes for **LEFT ARROW** and **CODE+LEFT ARROW** in the standard character set.

When the K1 keyboard is attached, there is no need for emulation. The operating system loads a default *null* emulation table to resolve internal pointers to an emulation data block.

Decoding

The decoding table is a keyboard supporting table that can optionally be included in the translation data block when it is created. The table specifies the unencoded keystrokes responsible for producing a particular character code. (A menu provided with the **Create Keyboard Data Block** command allows the user to include or exclude this table. See the *CTOS Executive Reference* for details.)

Context Manager, for example, uses decoding to accomplish its Cut And Paste function. Using the decoding table, it decodes the stream of character codes specified in the first context and copies the unencoded values to its memory. Then it calls `WriteKbdBuffer` to write the unencoded values into the type-ahead buffer of the second context. From the type-ahead buffer, the unencoded values are passed directly to the application reading the keyboard in unencoded mode. The values are translated to displayable characters before being passed to the character mode application.

Comparing Data Blocks

An emulation data block is organized in a similar fashion to a translation data block. (See “Translation Data Block Organization.”)

Like the translation data block, the emulation data block contains

- Offsets to the emulation tables
- Supporting tables to determine which emulation table to use and to obtain other details about the keyboard state

Note: *Emulation data blocks must be placed in writable segments only because the operating system needs to write internal state information to them.*

Figure 11-4 shows the translation data block and emulation data block memory layouts. In each data block, the structures listed above the dashed line are checksummed to ensure they are not corrupted. The supporting tables and the translation and emulation tables listed below the dashed line can be modified by the programmer customizing tables for application-specific or system-wide use. Note that there are a few different supporting tables for translation and emulation. The emulation LEDs table, for example, only occurs in an emulation data block.

Figure 11-4. Comparing Data Blocks

Translation Data Block Header	Emulation Data Block Header
Translation Table Descriptor Array	Emulation Table Descriptor Array
Multibyte String Offsets	Multibyte String Offsets
Decoding Offsets	...
...	...
Decoding	Multibyte Strings
Multibyte Strings	Special Keys
Special Keys	Chords
Chords	...
Control Cords	Conditions
Conditions	Diacritic Masks
Diacritic Masks	Diacritics
Diacritics	Multibyte Masks
Multibyte Masks	...
Command Masks	Emulation LEDs
...	Allowed States
Allowed States	...
Repeat Attributes	Used Table Masks
...	Emulation Table 0
Translation Table 0	.
.	.
.	.
Translation Table n	Emulation Table n

557.11-4

Emulating

Emulation maps the chord state and raw unencoded values on one keyboard (the source keyboard) to the chord state and unencoded values necessary to produce the same character on a different keyboard (the target keyboard). Some examples of emulation are described below. Also see “Redefining Keys.”

Emulation Examples

If, for example, a PC-style keyboard is to emulate a K1 keyboard, an emulation data block for a PC-style source keyboard to a K1 target must be loaded into memory. The emulation data provides the equivalent K1 chord state and key post value for each PC-style chord state and key post value. A K1 translation data block also must be available in memory to translate the K1 unencoded values resulting from emulation to the corresponding character codes. Note that the same K1 translation data block can be used with any emulation data block that emulates K1 hardware.

On the K1 keyboard, the shifted state of the 8 key on the key pad displays the asterisk (*). The key post value for the 8 key is 38h. On a PC-style keyboard the asterisk key on the numeric pad is displayed by the unshifted state of key post 7Eh.

To emulate the K1 asterisk key from a PC-style keyboard, the emulation data block must contain an emulation table for the unshifted state of PC-style keys. When the keyboard process uses the PC-style key post value 7Eh as the index into this table to obtain the corresponding K1 value, it should find the following entry:

01:38

The values 01h and 38h are the corresponding K1 keyboard chord state and key post values, respectively, for the unshifted state of PC-style asterisk key. The value 01 usually is used to represent **LEFT SHIFT**. (For details, see “Chord Information Table” in “System Structures” in the *CTOS Procedural Interface Reference Manual*. The chord state word identifies the current chord keys pressed.)

The K1 translation data block translates the resulting K1 chord state and key post value to the character code. Among the translation tables in this data block, there must be a table for the shifted state of K1 keys. When the keyboard process uses the K1 key post value 38h as the index into this table to obtain the K1 character code, it should find the following entry:

2A

The value 2Ah is the character code for the asterisk (*) on a K1 keyboard.

Note that emulation of a key causes the unencoded values for the emulated keyboard to be placed in the application type-ahead buffer. Applications reading unencoded values from the PC-style keyboard would receive the corresponding K1 unencoded values when the PC-style keyboard key is pressed and released.

When asterisk key is pressed on the PC-style keyboard, the application would receive the following K1 unencoded values in its type-ahead buffer:

Value	Meaning
48h	(K1 LEFT SHIFT)
38h	(K1 s key)

Upon release of the PC-style asterisk key, the application would receive the following K1 unencoded values:

Value	Meaning
B8h	(K1 s key)
C8h	(K1 LEFT SHIFT)

Many existing unencoded applications depend on K1 keyboard values. To support these programs on systems using non-K1 keyboards, it is necessary that the K1 translation data block and the appropriate emulation data blocks be present in the system keyboard file. By default, the system keyboard file contains the necessary keyboard data blocks for each keyboard supported.

Redefining Keys

Note: *Although, by default, emulation and translation data blocks are used to emulate and translate the values for the keyboard applications expect is being used, an application can use the data blocks to define keys for any unique purpose.*

Keyboard data blocks are designed such that emulated key values are separate from raw values. This design allows the user some flexibility in defining keys in ways other than the expected. It also explains how the operating system is able to support differences (such as the numbers of chord keys supported) between source and target keyboards. The arrangement allows emulation and translation data blocks to be built independently.

Take the chords supporting table, for example. Both the emulation and the translation data block versions of this table are the same format: each contains fields for the same hardware information (describing LEDs, toggles, and so forth). Separate fields define the raw key value and the emulated value. (For the format of the chords supporting table, see “System Structures” in the *CTOS Procedural Interface Reference Manual*.) This setup allows the keyboard customizer to redefine the emulated value for a raw key by overwriting the expected value with a different one. (See “Customizing the Keyboard Data Blocks” for ways to change data blocks.)

NUM LOCK, for example, is an extra chord on the PC-style keyboard not found on a K1. If the applications assume a K1 keyboard is being used, NUM LOCK can be assigned an emulated value from among the unused ranges of K1 keyboard codes. (See Appendix C in the *CTOS Procedural Interface Reference Manual* for a list of the unused K1 keyboard codes.) Then, the emulation table for the condition, NUM LOCK pressed, can redefine NUM LOCK as a different K1 key or key combination. Note that a chord key must be represented in the allowed state in the translation data block for its value to be passed though to the application type-ahead buffer. (For additional information, see “Selecting the Translation Table to Use.”)

Note: *There can be different numbers of chords (not shared in common) in an emulation and a translation data block. However, the chord bit mask positions of common chord keys in each data block must be the same. If, for example, bit mask position 1 defines the state of the Left Shift key in the emulation data block, Left Shift also must be bit mask position 1 in the translation data block.*

Emulating LEDs

The emulation LEDs supporting table maps LEDs from the source keyboard to the appropriate key on the target keyboard. This table essentially controls the commands sent out to the target keyboard hardware to turn LEDs on or off. (For further information, see “Emulation LEDs Table” in the *CTOS Procedural Interface Reference Manual*.)

Customizing the Keyboard Data Blocks

You can create or make changes to keyboard data blocks dynamically through system calls or by using the Keyboard Customizing utility packaged with the Standard Software Development Utilities. The customization tool consists of the following Executive commands:

- **Create Keyboard Data Block**
- **Convert Nls.sys**
- **Convert Sys.keys**

See the *CTOS Executive Reference* for details on how to use these commands.

The recommended way to create or modify keyboard data blocks is by using the **Create Keyboard Data Block** command. Using a text editor, you can open and edit the text file versions of the data blocks. Internal comments are included to assist you with making changes.

From the text file, you can create two different output files: an object module or a binary file. You can also edit data blocks to create an entirely new system keyboard file. By running the customized text files through the utility, you can create all the object modules you will need to build a new system keyboard file. (See “Building a New System Keyboard File.”)

Customizing the System Keyboard Data Blocks

Using the **Convert Nls.sys** or **Convert Sys.keys** command, you can use an existing *Nls.sys* or *Sys.keys* file containing customizations you made to create an object module or binary file containing the corresponding keyboard data blocks, or you can build a new system keyboard data block. Using the conversion commands may not result in exactly what the keyboard needs (for example converting a K1 *Nls.sys* file will not emulate K5 keys). By using a text editor, however, you can edit the results of conversion.

See the *CTOS Programming Utilities Reference Manual: Customization* for details on how to use the keyboard file conversion commands.

Building a New System Keyboard File

For each keyboard type, the Keyboard Customizing utility provides the appropriate keyboard data blocks in text file and object module format.

To create a customized version of the system keyboard file *NlsKbd.sys*, you can selectively link the object modules for the keyboard data blocks you want included in your version of the file.

Customizing Application-Specific Data Blocks

The recommended way to create or edit tables for use in creating an application-specific keyboard data block is by editing the existing keyboard data block text files.

By entering these text files as input to the **Create Keyboard Data Block** command, you can optionally create an output file in object module format or in binary format. If your purpose is to use a table of special characters with one application, you can link the object module you create with the application. If you create a binary file, it can be dynamically accessed by several applications in memory. Alternately, an application can select from several binary files according to the attached hardware and the application requirements.

Dynamically Customizing Data Blocks for Application Use

Your application can customize copies of the system keyboard data blocks for its own use by making programmatic calls to the operating system to obtain copies of the keyboard data blocks to modify. Once modified, the data blocks can then be posted for that application partition and used by the operating system while the application is running.

If Your Application Uses NLS Keyboard Tables

If your application uses a copy of the old NLS keyboard tables for its own use, it is valid to use the GetPStructure operation to obtain a pointer to the tables. (For details on GetPStructure, see the *CTOS Procedural Interface Reference Manual*.) However, an application must not expect the keyboard process to do anything with the NLS tables. Changes your application makes are effective purely on an individual basis. Note that *Sys.keys* files are not supported in this same way.

By default, GetPStructure with structure code 270 returns a pointer to NLS keyboard mapping table number 15 if both keyboard mapping tables (that is, NLS table number 0 and table number 15) are present in the file, *Nls.sys*. This default can be changed through a system configuration option. (See the *CTOS System Administration Guide* for details on system configuration options. For details on the NLS keyboard mapping tables, see “Keyboard Mapping” in the section entitled “Native Language Support.”)

Modifying Data Blocks

In most cases, it is easier to edit a keyboard data block text file accompanying the Keyboard Customizing utility than to make modifications dynamically. However, you can make dynamic modifications to tables in a data block your application has copied to local memory.

Figure 11-5 shows the format of *NlsKbd.sys*. Knowing the format of the table of contents (TOC) in this file will help you locate keyboard data blocks. To be able to interpret the keyboard data block contents, you must become familiar with the structure formats. (For details on the keyboard structures, see “System Structures” in the *CTOS Procedural Interface Reference Manual*.)

Note: *With the exception of data block headers and offset fields (which are checksummed), all other keyboard data block fields can be changed.*

The *NlsKbd.sys* TOC, starting at logical file address (lfa) 200h from the start of the run file, contains a 4 word entry for each keyboard data block.

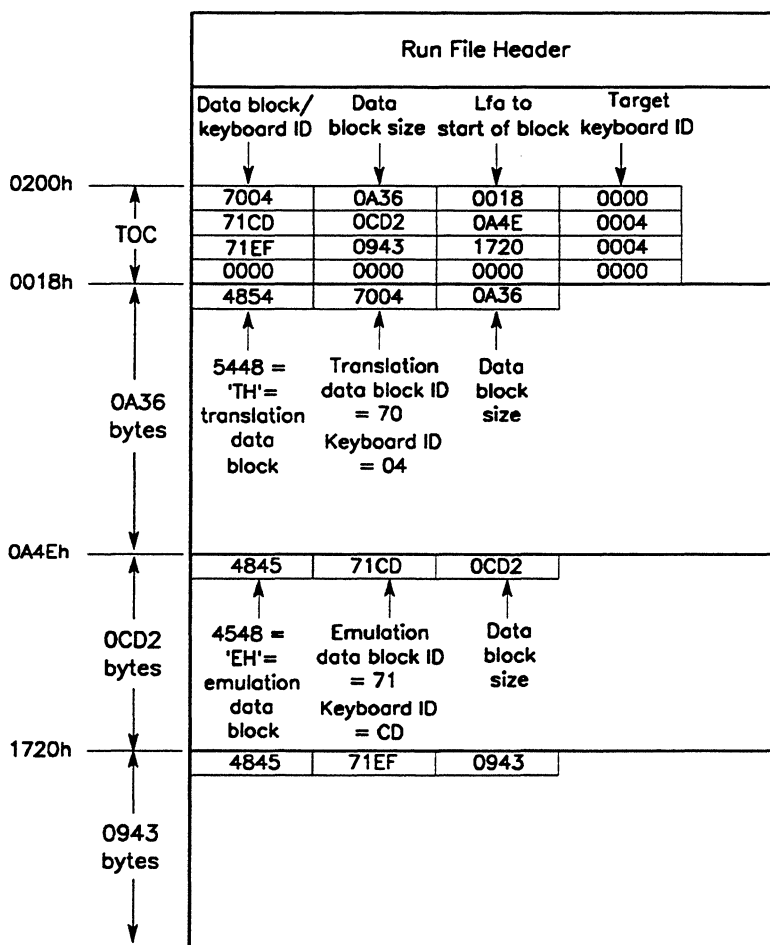
The first word of a TOC entry contains the keyboard ID (low-order byte) and the data block ID (high-order byte). The ID of a translation data block is 70h; an emulation data block ID is 71h.

The second and third words of the entry contain the data block size in bytes and the lfa of the data block from the start of the TOC.

The fourth word of the emulation data block entry contains the target keyboard ID. It contains zeros for a translation data block. Note that the last TOC entry is all zeros.

Figure 11-5 shows two emulation data blocks with keyboard IDs CDh and EFh emulating the target keyboard with ID 04h.

Figure 11-5. NlsKbd.sys Format



557.11-5

Note: The offset fields within a data block are offsets from the start of the individual data blocks, not lfes from the start of the file, NlsKbd.sys.

Posting Data Blocks

An application can call the `PostKbdTable` operation to post a locally based data block to the operating system. Posting causes the pointers in the partition keyboard information structure to point to the local data blocks rather than to the data blocks in the system keyboard file. (For details on this structure, see “System Structures” in the *CTOS Procedural Interface Reference Manual*.)

`PostKbdTable` can also be called to perform the opposite function: to ‘unpost’ a data block. Unposting switches from using application-specific keyboard data blocks to using the system data blocks.

Reading Keyboard Data Blocks

The paragraphs that follow describe four ways an application can read keyboard data blocks. The first three ways read the blocks into application memory. The fourth explains how to access a data block in an object module linked with an application. These methods are listed below:

- Reading a system data block in *NlsKbd.sys* into local memory using the `ReadOsKbdTable` operation
- Reading a system data block in *NlsKbd.sys* into local memory using byte streams
- Reading a binary data block file into local memory using byte streams
- Reading an object module linked with an application containing one or more data blocks

For details on the parameters to the CTOS operations used to describe these methods, see the *CTOS Procedural Interface Reference Manual*.

Note: *Emulation data blocks must be placed in writable segments only because the operating system needs to write internal state information to them.*

Using ReadOsKbdTable

You need to read a system keyboard data block into local memory before you customize it to post when your application is running. To determine how much memory to allocate for the block, an application can call the ReadOsKbdTable operation. The operation returns the size of the specified system data block when provided a data block size of 0 or a size smaller than the data block.

From this information, the application can allocate the appropriate amount of short-lived memory. By calling ReadOsKbdTable a second time and specifying the table size, the application can read a copy of the system data block into the memory allocated.

Using Byte Streams

See Figure 11-5 for the format of *NlsKbd.sys*. The following steps summarize how an application can approach reading a system data block from *NlsKbd.sys* into local memory using byte streams.

1. In your application, define an array of words to contain a table of contents (TOC) entry, for example
WORD prgbToc [4]
2. To read the file *NlsKbd.sys* as a byte stream, open it with the OpenByteStream operation, providing the file specification of *NlsKbd.sys*, for example
OpenByteStream (pBswa, pbNlsKbd.sys, cbNlsKbd.sys, ...)
3. Set the lfa to the start of the TOC, for example
SetBsLfa (pBswa, lfaTocStart)
4. Your application can loop through TOC entries, reading one entry at a time into the TOC array to find the specified data block. Using the keyboard ID, you can locate the specified data block from among the data block IDs and target IDs in the TOC.
5. Use the size of the data block (*cbBlock*) to gauge how much short-lived memory to allocate for it. Then allocate that much memory.

6. Set the lfa to the start of the data block (the lfa in the TOC is from the start of the TOC, not the start of the file); then read the data block into memory as a record, for example

Call *ReadBsRecord* (*pBswa*, *pDataBlock*, *cbBlock*, *pcbRead*)

7. Post the local data block so the operating system will use it while the application is running, for example

Call *PostKbdTable* (0, *pDataBlock*, *cbBlock*)

For details on the parameters to the CTOS operations used to describe this method, see the *CTOS Procedural Interface Reference Manual*.

Binary Data Block File to Local Memory

Using byte streams to read a binary data block file into local memory works much the same as reading a data block from *NlsKbd.sys* using byte streams. (See “Using Byte Streams.”) In either case, your application needs to allocate local memory for the block. Once memory is allocated, the data block can be read in as a byte stream record.

The difference lies in the data block format being read. Unlike the *NlsKbd.sys* file, a binary data block file contains no table of contents. The file starts immediately with the first byte of the data block.

All your program needs to do is to read the first three words in the data block header into an array. (See “System Structures” in the *CTOS Procedural Interface Reference Manual* for the format of a translation or emulation data block header.) The third word contains the size of the data block. Once your application determines the size, it can allocate the appropriate amount of short-lived memory for the data block. Then it can read the data block as a byte stream record into the specified memory area.

The application posts the resulting data block copy using the *PostKbdTable* operation.

Data Block in an Object Module

You can use the **Create Keyboard Data Block** command to create object module output from a data block text file. If you link a group of these object modules together, the Linker will create a module with a table of contents at the beginning. To facilitate finding a block when the module contains several objects, the command creates a public procedure name located at the first byte of the data block in each object module. Your program can use this name as a point of reference to locate the block.

To find a specified block dynamically, define a public variable with the procedure name, for example,

```
extern char KbdDataBlock
```

Comparing Methods of Reading Data Blocks

Each of the methods for reading data blocks has its advantages and disadvantages. (See “Reading Keyboard Data Blocks” for details.)

The advantage of first method (reading a system data block using `ReadOsKbdTable`) is that the operating system selects the right data block to correspond to the attached keyboard. Your application doesn't need to find it. The advantage of this method and the second (reading a system data block using byte streams) is that major customizations (such as nationalization) may already have been done to the system keyboard files. All you may need to do is make minor changes (to data block areas not checksummed) to meet your application requirements. (See “Modifying Data Blocks.”)

The third method (reading a binary data block file) requires determining the correct path information. However, it has the advantage of not taking up memory in your application until the data block is used. Also, a data block starts at the very beginning of a binary file. Using the **Create Keyboard Data Block** command, you can append the contents of binary files so that the file can contain several data blocks. An application can check the data block size in the header of the previous data block to locate the next and subsequent data blocks in the file.

The fourth method (reading an object module containing one or more data blocks) avoids problems with establishing the correct path. However, the object module consumes application memory.

Application Customization Examples

The following features can be defined by editing keyboard data blocks.

- Multibyte strings
- Diacritics
- Chords that toggle

Examples of ways to use these features are described in the paragraphs that follow.

Multibyte Strings

In the translation and emulation data blocks, a supporting table defines multibyte strings. Using multibyte strings, pressing a single key can generate a character string. This feature allows you to take advantage of little used keys to generate frequently typed character sequences. If, for example, you do not use the 0 on the numeric pad when the CODE key is down, but you frequently need to enter the name of a system file for a different path, you can define CODE and 0 to generate a character string for the system volume and directory, for example

[Sys]<Sys>

Diacritics

There is also a supporting table defining diacritic key pairs. Diacritic key handling is useful for displaying characters with diacritical marks, such as the German *ä* with an umlaut. The first key of a diacritical key pair enables diacritical mode; the second key displays the diacritical result. Any character codes can be assigned diacritical key handling.

Chords that Toggle

To define a toggle chord, you have to define the chord to be a new chord type (type toggle) in the chords supporting table. You will also need to update the conditions and the allowed states supporting tables to reflect this chord type. (See “Selecting the Translation Table to Use,” for details on these supporting tables.) Then you can define the chord to function as a toggle in your application.

For example, suppose you were writing a program that quotes original Greek text. If your program also uses English strings, you can define a chord to toggle between the English and Greek character sets (presuming fonts support all the English and Greek characters). To display the Greek characters, you would need to create additional translation tables in the translation data block to define the Greek character set. (See “Customizing the Keyboard Data Blocks.” You also must have Greek display fonts.) Then the keyboard process can select the appropriate translation table to use based on the conditions and allowed states. (See “Selecting the Translation Table to Use,” for details.)

System Profile Keyboard

The *system profile keyboard* is the keyboard emulated on an operating system. The default translation table in *NlsKbd.sys* matches the system profile keyboard. All default emulation tables contain the system profile keyboard ID in the *targetId* field of the data block header. (See “System Structures” in the *CTOS Procedural Interface Reference Manual* for the format of the emulation data block header.)

Note that an application can have a profile keyboard of its own that may not match the system keyboard profile. (See “Application Profile Keyboard,” for details.)

An example of a system profile keyboard is the default U.S. K1 keyboard. It is the profile keyboard in most U.S. operating systems. Typically, unencoded applications that run on an operating system with U.S. defaults expect K1 keyboard values. Several different keyboards can be used, however, as long as each non-K1 keyboard is defined by the appropriate keyboard data blocks in the system keyboard file *NlsKbd.sys*. A K5 keyboard can be used, for example, if the system keyboard file contains a K5-to-K1 emulation table. The *targetId* field in the emulation data block header links each non-K1 keyboard to the K1 profile keyboard.

With these links in place, one K1 translation table can be used to translate all the K1 unencoded values to the appropriate character codes. (See “Matching Keyboards to Data Blocks.”)

Matching Keyboards to Data Blocks

When the operating system is first initialized, it uses the following strategy to match the keyboard hardware to the appropriate keyboard data blocks:

1. In *NlsKbd.sys*, it looks for an emulation data block whose keyboard ID that matches the ID of the attached keyboard.
2. If it finds the appropriate emulation data block, it looks for the ID of the target keyboard in the emulation data block header; then it looks for a translation table with this same keyboard ID.
3. If it does not find an emulation data block with a keyboard ID matching the attached keyboard, it then looks for a translation data block whose keyboard ID matches the ID of the attached keyboard.
4. If doesn't find the appropriate translation data block, it uses the K1 translation data block linked into the operating system.

The point here is that the operating system will not load an emulation data block matching the attached hardware that emulates just any style keyboard. It either loads the emulation data block for the attached keyboard, or it looks for emulation and translation data blocks that match the attached keyboard hardware.

When the user changes keyboards the keyboard process ensures the correct data blocks are in memory for that keyboard. It checks the keyboard ID of the attached keyboard against the keyboard ID shown in the header of the emulation data block in memory. Then it uses the same procedure outlined in steps 1 through 3 above to match the keyboard to the appropriate data blocks. If there is no match, the data blocks for the previous keyboard continue to be used.

When an application posts its own data blocks, the operating system uses them instead of default system data blocks. (See "Posting Data Blocks," for details.) The application must ensure that the operating system can match keyboards to the data blocks it posts. Depending on its purpose, the application can use one of the following strategies:

- If it posts a translation data block that matches the ID of the attached hardware, it can post a null emulation data block. The null data block resolves emulation data block pointers when emulation is not needed.

- It can post data blocks with the same IDs as the current data blocks.
- It can post an emulation data block with a target ID of the keyboard it expects to be attached (the application profile keyboard) as long as it posts a translation data block with the same ID. (See “Application Keyboard Profile,” for details.)

Application Profile Keyboard

An application profile keyboard is the keyboard the application expects to be attached. If the application profile keyboard does not match the system profile keyboard, the system can be configured to support it. (See “Multiple Keyboard Profiles,” for details.) This allows applications to run without change on a system with a different profile.

Multiple Keyboard Profiles

The system configuration option `:KbdProfile:` allows a system to support more than one system profile keyboard.

The configuration option `:KbdTables:` allows the user to define the amount of memory required for loadable keyboard data blocks and the maximum number to be loaded. At system initialization, the operating system will reserve the amount of memory that is the greater of the following two values: the amount of memory necessary to load the largest translation data block and emulation data block into memory or the amount of memory specified by the size subparameter to the `:KbdTables:` configuration option. The memory reserved cannot be increased later.

(For system configuration file option details, see the *CTOS System Administration Guide*.)

To set up an application to use a profile keyboard other than the default system profile keyboard, the **Version** command can be run on the application run file. (For details, see the **Version** command in the *CTOS Executive Reference Manual*.) This feature allows applications already written to expect the values of its own profile keyboard to run without change on a system with a different profile keyboard. If the system configuration option :KbdTables: was used to reserve enough space for the data blocks and if the data blocks are contained in *NlsKbd.sys*, the operating system will load them on behalf of the application.

Mapping Keyboard IDs

A keyboard ID can be mapped to the ID of a different keyboard using the system configuration option :MapKeyboardId: or by an application dynamically calling SetKeyboardOptions. (For details on system configuration file options, see the *CTOS System Administration Guide*. SetKeyboardOptions is supported on protected mode operating systems only and is described in the *CTOS Procedural Interface Reference Manual*.)

Most typically, this feature would be used for keyboard types with only one keyboard ID (for example, the K1 or the PC-style keyboard). Several keyboards of the same type could be customized to different native languages and assigned different keyboard IDs through keyboard mapping. With different IDs, each keyboard would be associated with its own unique set of translation and emulation data blocks. Conveniently, these customized keyboards could be packaged in the same *NlsKbd.sys* while being controlled by individual system configuration options.

Setting Keyboard Options

On protected mode operating systems, an application can dynamically set a number of keyboard options using the `SetKeyboardOptions` operation. (The options are available to real mode systems through the system configuration file. See the *CTOS System Administration Guide* for details.) Some keyboard options provide support for persons with disabilities. For example, keys can remain in effect if pressed until the user presses a key they affect. Also, keys can be defined to repeat at slower rates or to not repeat at all. `SetKeyboardOptions` also allows your application to map keyboard IDs. (See “Mapping Keyboard IDs” for details on this topic.)

Sticky Keys

By using the `SetKeyboardOptions` operation, an application can dynamically define chord keys and `ACTION` as sticky keys. *Sticky keys* are pressed and remain in effect until a key they affect is pressed.

For example, in OFIS Document Designer, pressing the key sequence--`CODE`, `SHIFT`, and `P`--highlights a document. By defining `CODE` and `SHIFT` as sticky, you press and release each key, one at a time. Then, press `P`. Doing so highlights the document.

Note that this feature works for one character at a time. `P` cannot be pressed again without first pressing `CODE` and `SHIFT`. Doing so would only display the lowercase character `p` to the video device.

Using this feature, all non-toggle chords are sticky if any one of them is: they cannot be defined separately. Furthermore, if chords are sticky, `ACTION` also sticks but not vice versa: if `ACTION` is sticky, this does not mean chords are.

Character Repeating

The rate at which a character repeats also can be slowed or stopped dynamically by using the `SetKeyboardOptions` operation. Your program can repeat strings, diacritics, and chords as well as single characters.

System Input Process

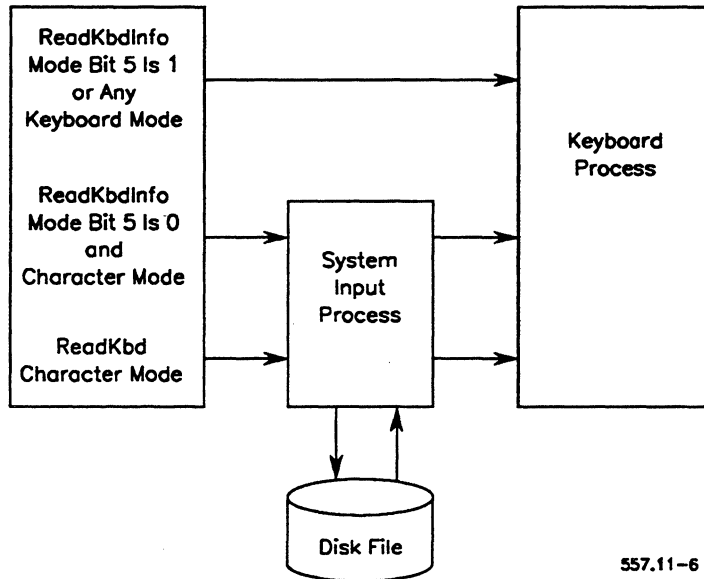
The *system input process* acts as a filter of requests to read the keyboard in character mode. (For details on filters, see “Filters.”)

When the system input process receives one of these requests from an application, it can operate in one of three modes: normal, recording, and playback (submit). Typically the modes are activated by an Executive command but they also can be activated dynamically when an application calls the *SetSysInMode* operation and specifies a mode. The modes are described briefly below:

- In *normal mode*, the system input process takes no special actions on the request. It simply forwards the request to the keyboard process, intercepts the keyboard process response, and forwards the response to the application.
- In *recording mode*, the system input process performs the same functions except, before forwarding the response to the application, it writes a copy of the keyboard input to a recording file.
- In *playback mode*, the system input process intercepts the request for keyboard input. The input is read from a specified file instead of from the keyboard, and the data is returned to the application as if it had come from the keyboard.

The system input process is shown in Figure 11-6. A *ReadKbdInfo* request is directed to the system input process when two conditions are met: the application is reading the keyboard in character mode, and bit 5 of the *ReadKbdInfo mode* parameter is 0. If the application is reading the keyboard in any other keyboard mode, the read request will go directly to the keyboard process instead. (See “Keyboard Modes” for details.) Setting bit 5 of the *ReadKbdInfo mode* parameter (to 1) also directs the read request to the keyboard process.

Figure 11-6. System Input Process



557.11-6

Playback Mode

In *playback mode* (also called submit mode), a ReadKbdInfo request is met by the system input process, which reads the data from a specified file instead of from the keyboard. Typically, playback mode is executed by the Executive **Submit** command. (For details on Submit, see the *CTOS Executive Reference Manual*.)

The file from which the data is read is a simple ASCII file. It could have been created by the system input process itself in recording mode. However, the file is often created using a text editor or the Executive Command File Editor. By convention, this file called a *submit file*.

Note: For backwards compatibility, ReadKbd is also supported to read submit files. (See Figure 11-6.) For ease in processing different multibyte character sets, however, new applications should use the ReadKbdInfo operation. (See "Reading the Keyboard," for details.)

When the `SetSysInMode` request is used to invoke playback mode, the caller must specify the file handle of the submit file to be read.

A submit file remains active until one of the following conditions is met:

- All characters in the file are read.
- A read-direct or end-of-file escape sequence is read. (See “Submit File Escape Sequences.”)
- `SetSysInMode` is called again.

Calling `ReadKbdInfo` operation while a submit file is active causes a 12-byte block of data to be read from the submit file and returned to the caller. (`ReadKbd`, on the other hand, only reads one byte at a time.) After all characters are read, the submit file is automatically closed. Subsequent calls to `ReadKbdInfo` cause characters to be read directly from the keyboard. Transition of input source from submit file to keyboard is totally transparent to the application program. If, however, an application needs to know whether a submit file is active, the `QueryKbdState` operation can be called to provide this information.

Note: *Applications must use `ReadKbdInfo` rather than `ReadInputEvent` to read submit files. Such applications, therefore, are not able to obtain Mouse or other nonkeyboard event types while a submit file is being read.*

An application can temporarily disable playback of a submit file by calling the `SetKbdUnencodedMode` operation or by reading a specific escape sequence. (For details on escape sequences, see “Submit File Escape Sequences.”)

The system input process will not read from a submit file if the requesting application is reading the keyboard in any keyboard mode other than character mode. (See “Keyboard Modes” for details.) To reactivate a submit file, however, the application can call `SetKbdUnencodedMode` to change the keyboard mode back to character mode. Subsequent characters thus are read from the submit file (provided bit 5 of the `ReadKbdInfo mode` parameter is 0).

Regardless of whether a submit file is active, calling `ReadKbdInfo` while bit 5 of the *mode* parameter is set (or calling the `ReadKbdDirect` operation) will cause the operating system to forward the request directly to the keyboard process.

Note: *ReadKbdDirect* is supported for backwards compatibility only. New applications are encouraged to use *ReadKbdInfo*.

Say, for example, the Executive is set up to pause between pages, and playback of a submit file requires that video byte streams display more than one screen of data. After displaying a screen of data, video byte streams must send the following message to the video to obtain a response from the keyboard user:

Press NEXT PAGE or SCROLL UP to continue

To read the response entered by the user at the keyboard, video byte streams calls `ReadKbdInfo` (with bit 5 of the *mode* parameter set).

Recording Mode

In *recording mode*, all characters typed at the keyboard and read in character mode by `ReadKbdInfo` (or `ReadKbd`) are written to a *recording file*, in addition to being returned to the caller. (Note that **Action** keys are not recorded.)

A recording file can be used later as a submit file to repeat the same sequence of input characters. A recording file and a submit file, however, cannot be activated at the same time.

Submit File Escape Sequences

Certain sequences of characters (*escape sequences*) invoke special functions when read from a submit file. A *submit file escape sequence* consists of two or three characters.

- The first character of the escape sequence is the character code 03h (cent sign), which indicates the presence of an escape sequence.
- The second is a code to identify the special function.
- The third character, if present, is an argument to the special function.

The permitted codes are shown in Table 11-2. The **Submit** command supports additional escape sequences as well. For details on creating submit files, see the description of “Submit” in the *CTOS Executive Reference Manual*. The description includes several examples of how to use escape sequences to obtain user input for Executive command line parameters as a submit file is being played back.

Table 11-2. Submit File Escape Sequence Permitted Codes

Message character	Code	Function
cent sign	03h	A 2-character escape sequence that represents the character code 03h. Since 03h (cent sign) is used to introduce escape sequences, this escape sequence (that is, two consecutive cent signs) is the only way to represent the cent sign in the submit file.
1	31h	A 3-character, read-direct escape sequence. (See “Read-Direct Escape Sequences.”)
2	32h	An end-of-file escape sequence. When this two-character escape sequence is read during a ReadKbdInfo operation, it closes the submit file. The current and subsequent ReadKbdInfo operations read characters directly from the keyboard. This escape sequence is meaningful only in submit files created using the Editor. It is not meaningful in submit files created as recording files.)
3	33h	The previous request during the submit file recording was ReadKbd. The current request is ReadKbdInfo.
4	34h	The previous request during the submit file recording was ReadKbdInfo. The current request is ReadKbd.

Caution

Unlike the ReadKbd operation, which returns a 1-byte character code with every keyboard read, ReadKbdInfo returns a 12-byte record. If you are editing a submit file, you should avoid altering any data between the submit file message characters (cent sign) 3 and (cent sign) 4 unless you fully understand the 12-byte record returned by ReadKbdInfo. For the format of the record ReadKbdInfo returns, see the CTOS Procedural Interface Reference Manual.

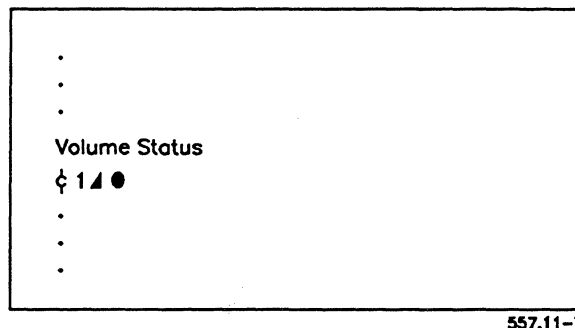
Read-Direct Escape Sequences

The *read-direct escape sequence* is a 3-character submit file escape sequence that causes ReadKbdInfo to read characters directly from the keyboard until a specified key is pressed. The third byte of the escape sequence specifies the key that is to terminate input from the keyboard. When the specified key is pressed, its key post value is not returned to the program. Rather, the current and all subsequent ReadKbdInfo operations read characters from the submit file (unless another escape sequence redirects the input source).

For example, it is frequently useful to have the user enter data into an Executive command form during the playback of a submit file. Such a submit file must include a read-direct escape sequence in the command field where the user is to enter the data.

(See Figure 11-7.) The figure shows a portion of a submit file. The first field (Volume name) in the **Volume Status** command requires user entered data. When the system input process reads the escape sequence cent sign 1 MARK (triangular display character) from the submit file, it leaves the cursor blinking in the leftmost character position of the command field. (See the *CTOS Executive Reference Manual* for details on the Volume Status command fields.)

Figure 11-7. Escape Sequence for Entering User Data



557.11-7

This prompts the user to enter the appropriate data into the field. Previous instructions in the submit file should tell the user to press **MARK** when done entering data. As soon as the user presses **MARK**, submit file execution resumes. The first character to be read in the submit file in Figure 11-7 is the **GO** character (displayed as a filled-in bullet). **GO** executes the **Volume Status** command.

***Note:** If you are recording and you press **ACTION+FINISH**, the recording file is closed gracefully so that it can be played back.*

Action Key

ACTION is a special kind of control key. By default, the key doesn't have an associated character code. It is processed specially, even in unencoded mode. (If indeed **ACTION** does have a translatable character code, it does get passed through to the type-ahead buffer but the keys pressed in combination with it do not.)

The values of keys pressed with **ACTION** do not get placed in the type-ahead buffer. An application must call the **ReadActionKbd** or **ReadActionCode** operation to obtain the values.

The operating system processes key combinations that include **ACTION** independently of calls made to **ReadKbdInfo**. Such key combinations are not affected by unencoded values stored in the type-ahead buffer. These key combinations and their functions are shown in Table 11-3.

Table 11-3. Action Key Combinations

Key Combination	Function
Action+Delete	Clears the type-ahead buffer.
Action+Overtyp	Blanks out the screen. ACTION+OVERTYPE does not affect any ongoing activity, but simply makes the screen blank. To reactivate the video display, press any nonediting key, such as SHIFT or CODE .
Action+Finish	Terminates the execution of the current program and invokes the exit run file. The DisableActionFinish operation disables this feature.
Action+A	Invokes the Debugger in simple mode. (See the <i>CTOS Debugger User's Guide</i> for details.)
Action+B	Invokes the Debugger in multiprocess mode. (See your debugger manual for details.)

Key combinations that include **ACTION** are available for program interpretation. Pressing **ACTION** in conjunction with any other key causes the keyboard code for that key to be stored in keyboard management memory. The keyboard code (also called an *action code* because of its association with **ACTION**) can be obtained by calling **ReadActionCode** or **ReadActionKbd**. Calling either of these operations avoids changing modes to obtain this information, thereby allowing the type-ahead buffer to continue while the program tests for special user intervention.

The BASIC interpreter, for example, uses **ACTION+CANCEL** to interrupt computation without interfering with type-ahead. The Context Manager uses **ACTION+GO**, **ACTION+NEXT**, and **ACTION+F1** through **ACTION+F10** for switching from one context (user number) to another.

ReadActionKbd can be called to determine immediately if an **ACTION** key sequence is used. Typically, **ReadActionKbd** is used asynchronously. (For details on the asynchronous use of requests, see the section entitled "Interprocess Communication.")

At Program Termination

When an application program terminates (because of the Chain, Exit, or ErrorExit operations, or ACTION+FINISH), termination has the following effects on keyboard management:

- If the keyboard was in unencoded mode, it is reset to character mode, and the content of the type-ahead buffer is discarded.
- The ACTION+FINISH feature is reenabled.
- The action code, if any, is discarded.
- If the program had posted its own keyboard data blocks, the operating system resets back to the default data blocks.

If the program terminates abnormally (because of the Chain or ErrorExit operations with a nonzero status code, or ACTION+FINISH), termination has the following additional effects:

- The content of the type-ahead buffer is discarded.
- The submit or recording file is closed.

Termination of the program does not affect the keyboard LEDs. The Executive, however, resets the LEDs when it is loaded.

Keyboard and Video Independence

Keyboard management does not automatically echo characters to the video device. A program can assign various functions to each character and can select whether or not to echo the characters. Keyboard management attaches no special significance to keys such as FINISH, HELP, RETURN, or DELETE. ACTION is the only key with special significance.

I-Bus Device Management

I-Bus device management allows you to write loadable I-Bus drivers to interface with I-Bus devices. I-Bus handlers process input from devices such as magnetic card readers or light pens that are attached to the I-Bus.

The `SetIBusHandler` operation installs a loadable I-Bus driver. As part of the installation process, the driver is registered to process data from a specified I-Bus device. When the specified device requests a service, such as a read of data, the operating system passes the read request onto the device handler.

The device handler processes the request and calls the `WriteIBusEvent` operation to return input event codes and keyboard character codes to the operating system. The operating system, in turn, queues the events in the current keyboard owner's event queue.

Note: *WriteKbdBuffer should be used instead of WriteIBusEvent for any input event type that must be passed to a specific application.*

The `ResetIBusHandler` operation disconnects an I-Bus driver from a specified device.

The operating system can identify I-Bus devices assuming they follow I-Bus protocol. Each device is associated with an I-Bus device ID, which is placed in an I-Bus table. [See the section entitled "X-Bus Management," for details on how the operating system system generates I-Bus (and X-Bus) IDs and how an application can obtain an ID for a given device. For a list of all the I-Bus IDs currently used, see Appendix G, "X-Bus and I-Bus Module IDs," in the *CTOS Procedural Interface Reference Manual*.]

The `WriteIBusDevice` operation can be used to output a byte string to an I-Bus device.

If an I-Bus device such as a mouse pointing device is used concurrently with the keyboard, `ReadInputEvent` should be used instead of `ReadKbdInfo` to read input events. The operation returns information generated by events such as the following:

- Pressing a mouse button
- Pressing a keyboard key
- Moving out of a motion rectangle

In addition, the operation returns all the details about a keyboard event that are returned by the `ReadKbdInfo` operation. (For details, see "Using `ReadKbdInfo`.")

Keyboard and I-Bus Management Operations

The keyboard management operations described below are categorized by use. Operations are arranged alphabetically in each group. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Basic

Beep

Activates one or more audio tones for 0.3 second and allows the caller to specify the tone.

GetKeyboardId

Returns the I-Bus ID of the keyboard attached.

GetKbdId

Returns the keyboard ID to the caller. (This operation is for backwards compatibility only. New applications should use *GetKeyboardId*.)

QueryKbdLeds

Returns the status (on or off) of the 8 keyboard LEDs.

ReadKbd

Reads one character from the keyboard, or from a submit file if one is active. (This operation is for backwards compatibility only. New applications should use *ReadKbdInfo* or *ReadInputEvent*.)

ReadKbdDataDirect

Reads and returns one character code (or keyboard code, if in unencoded mode) from the keyboard regardless of whether a submit file is active. In addition, the *ReadKbdDataDirect* operation returns information about the keyboard state. (This operation is supported for backwards compatibility only. New applications should use *ReadKbdInfo*.)

ReadKbdDirect

Reads and returns one character code (or keyboard code, if in unencoded mode) from the keyboard regardless of whether a submit file is active.) This operation is supported for backwards compatibility only. New applications should use ReadKbdInfo.)

ReadKbdInfo

Returns status information about a keystroke in the form of a 12-byte keyboard event record. Status includes information such as whether the key is a chord key, if the event is an upstroke, the raw hardware value of the key, the character code (if one is defined), and the character length (one, two, or four bytes). In character mode ReadKbdInfo can read input from the keyboard or from a submit file depending on the value of bit 5 of the *mode* parameter.

SetKbdLed

Turns on or off 1 to 16 keyboard LEDs.

Advanced

CheckpointSysIn

Writes the contents of the current, partially filled, output buffer to the recording file if the system input process is in recording mode.

DisableActionFinish

Disables operating system interpretation of ACTION+FINISH.

KeyboardProfile

Returns the keyboard ID and type of the attached keyboard

PostKbdTable

Directs the operating system to use a keyboard data block stored in application partition memory when the application is running. The operation can also be called to revert back to the current operating system keyboard data block.

QueryKbdState

Returns the status of the keyboard and of the system input process to a structure provided by the program.

ReadActionCode

Returns the action code, if any, and resets the indication that an action code is available.

ReadActionKbd

Detects ACTION key sequences.

ReadOsKbdTable

Reads a copy of the current operating system keyboard data block to the specified area in application memory. If the buffer size provided is 0 or too small to contain the data block, the operation returns the size of the keyboard data block to the address provided.

SetKbdUnencodedMode

Selects the mode in which the keyboard is read.

SetKeyboardOptions

Allows the caller to specify keyboard configuration options such as loading a different keyboard data block, causing the keyboard to beep on toggle, changing the character repeat rate, and causing Action or chord keys to remain in effect when the keys are released.

SetSysInMode

Changes the state of the system input process.

WriteKbdBuffer

Writes the unencoded keystroke values used to generate character codes into the application type-ahead buffer.

I-BUS Management

ResetIBusHandler

Allows the caller to disconnect an I-Bus handler from a specific I-Bus device that the handler was registered with using *SetIBusHandler*.

SetIBusHandler

Allows the caller to register (with an I-Bus interrupt handler) a procedure for handling or processing data from a specific I-Bus device.

WriteIBusDevice

Writes a byte string to a specific I-Bus device.

WriteIBusEvent

Places I-Bus events (such as keystrokes and mouse movement) in the queue for the current type-ahead buffer as set by the *AssignKbd* operation. (*AssignKbd* is described in Chapter 35, "Partitions and Partition Management.")

Magnetic Card Reader

PurgeMcr

Flushes the operating system internal MCR buffer.

ReadMcr

Reads the MCR data according to various operation modes.

Section 12

File Management

What is File Management?

The file management system provides a hierarchical organization of disk file data by node, volume, directory, and file. The operating system automatically recognizes a volume when it is placed online (mounted). A file can have a 50 character file name, a 12 character password, and a file protection level. A file can be dynamically expanded and contracted without limit as long as it fits on one disk (up to 1 gigabyte). Concurrent access is controlled by read (shared), peek (shared), and modify (exclusive) access modes.

While providing convenience and reliability, the file management system supports the full throughput capability of the disk hardware. This includes reading or writing any 512 byte sector of an open file with one disk access, reading or writing up to 64K bytes (127 sectors) of an open file with one disk operation, overlapping I/O with process execution, and optimizing disk arm scheduling.

From a cluster workstation with a local file system, you can access local files as well as files located at the server.

Overview of File System Capabilities

The file system has features that contribute to its efficiency, reliability, and convenience.

Efficiency

File system efficiency is provided through the following methods:

- **Careful data placement:** The operating system places the volume control structures, which are resident on each volume, at locations that minimize disk arm movement.

The operating system brings the volume home block into memory when you place a volume online. In addition, it retains the most recently used directory and file information in memory.

- **Randomization (hashing) techniques:** The operating system uses randomization techniques for placing an entry in a directory sector and later for locating the entry. These techniques reduce the number of disk reads required to access directory information.
- **Disk caching:** Protected mode operating systems support caching of disk data in main memory. (For details on the disk cacher, see “How Caching Works,” in the *CTOS System Administration Guide*.) Disk caching minimizes the frequency of disk access, thereby increasing overall performance and throughput. Furthermore, hashing techniques augment the efficiency with which the disk cacher references the requested disk sectors. Provided a shared resource processor includes a protected mode board, the Remote Cache Service allows caching of disks attached to real mode boards as well.
- **File caching:** Protected mode cluster workstation operating systems support caching of server file data in the cluster workstation main memory. (For details on configuring the Agent file cache, see “Caching Files From the Server,” in the *CTOS System Administration Guide*.) This distributed file caching minimizes the frequency of cluster communication file access, thereby increasing overall cluster performance and throughput.

Reliability

Reliability is provided through the following features:

- **Duplication of two volume control structures:** the volume home block and the file header blocks.

This duplication ensures that damage to one copy of a volume control structure does not cause data loss.
- **Ordered updating of volume structures:** This ensures that the volume will not be corrupted by power failure, hardware malfunction, or software error.

- Multilevel (volume, directory, or file) password protection.
- Multiple file protection levels: A file protection level specifies the access allowed to a file when the program requesting access does not provide a valid volume, directory, or file password.
- Optional volume encryption: You can optionally encrypt the passwords of all files and directories created on a volume. Volume encryption ensures that a file cannot be opened without a valid password.

Convenience

Convenience is provided through the following means:

- Hierarchical organization of disk file data by node, volume, directory, and file.
- Long file names (up to 50 characters).
- Dynamic file length: You can determine the file length when you create the file, and you can change file length later.
- Removable file volumes (floppy disks).
- Automatic recognition of volumes placed online.
- Ability to open files in read (shared), peek (shared), and modify (exclusive) mode.
- Device independence: The device on which a file is located is transparent to you.

Structured File Access Methods

Structured file access methods augment the file management system by providing additional structured access to disk file data. The structured file access methods are

- The record sequential access method
- The direct access method
- The Indexed Sequential Access Method

Access to Local Files

A cluster workstation can have its own local file system. The *local file system* allows a cluster workstation to access files on its local disks as well as files on disks at the server. The operating system routes processing requests to either the local or server file system on the basis of file specifications or handles. (For details on routing requests, see “Interprocess Communication.”)

You can bootstrap a cluster workstation either from a file at the server or from the local file system. A cluster workstation bootstrapped from its local file system is a self-contained entity that accesses the server only for shared files. If a malfunction occurs at the server, the cluster workstation can continue to operate normally, provided all of the files you access are on your workstation’s local disks.

An application program can access a server file system in the same way the program accesses a standalone workstation’s local file system. A program that works on a standalone workstation will work correctly on a cluster workstation that accesses server files.

File Specifications

The file management system organizes disk file data hierarchically from top level to bottom level as follows:

- Node
- Volume
- Directory
- File
- Password (optional)

Each of these levels is described in the paragraphs that follow. For restrictions on volume, directory, and file naming, see the section on the file system in the *CTOS Executive User’s Guide*.

Node

A system connected to a network can access the files of other network *nodes*, subject to password protection. If the file you are requesting is not on your node, you must specify the different node when attempting to access the file.

A node name is a string of characters. It can have a maximum of 12 characters.

Volume

The files of the system are located on *volumes*. In the Executive, use the **Format Disk** command to format and initialize a volume. (For details on Format Disk, see the *CTOS Executive Reference Manual*.) You can protect a volume with a volume password and by volume encryption.

A floppy disk and the media sealed inside a hard disk drive are examples of volumes. A floppy disk is a removable volume.

Volume Name

A volume name is a string of characters. It can have a maximum of 12 characters.

System Volume

Sys is a mnemonic for the volume name of the disk from which the operating system was bootstrapped.

For example, in a hard disk system where the operating system was bootstrapped from hard disk drive 0, you can use *Sys* instead of its volume or device name.

In a cluster workstation without local disk storage, *Sys* is a synonym for the volume name of the disk on the server from which the workstation was bootstrapped.

/Sys signifies the volume name of the disk from which the server of the cluster was bootstrapped.

Scratch Volume

You can reference the volume on which scratch (temporary) files are placed either by its mnemonic *Scr* or by its real name. The volume to be used as the scratch volume (*Sys* by default) is determined at system build (*SysGen*). For protected mode, the scratch volume also can be determined by an entry in the system configuration file, *[Sys]<Sys>Config.sys*. (For details, see the *CTOS System Administration Guide*.)

Volume Control Structures

A volume contains several *volume control structures*: the volume home block, the file header blocks, and the master file directory, among others.

The volume home block is the root structure of information for a disk volume.

The file header block of each file contains information about that file and about the disk address and size of each of its disk extents. (A disk extent is one or more contiguous disk sectors.)

The master file directory contains an entry for each directory on the volume. The directories provide fast access to the file header block of a specific file. They do not, however, contain any information about the file that is not also contained in its file header block.

Volume home blocks (working and initial copies) and file header blocks (primary and secondary copies) each have duplicates on the volume for reliability.

The location on the volume of the volume home blocks, the file header blocks, and the other volume control structures minimizes disk arm movement and therefore maximizes efficiency. The file header blocks are located in a single area of the volume, the disk address and size of which are recorded in the working and initial copies of the volume home block. Volume control structures that the operating system accesses frequently, including the primary and secondary copies of the file header blocks, are located near the middle of the disk.

Directory

The files of a volume are divided into one or more directories. A *directory* is a collection of related files on one volume. The maximum number of directories that you can create on a volume depends on the size of the master file directory, which you can specify when you initialize the volume. The maximum number of files that you can create in a directory depends on two factors:

- The directory size that you specified when you created the directory
- The length of all names of all files in that directory

A directory can be protected by a directory password. To create files in the directory with the CreateFile operation, you need the directory password. No password is required, however, to read file names in the directory using the ReadDirSector operation.

You can create a directory with the CreateDir operation and delete it with the DeleteDir operation.

A directory name is a string of characters. It can have a maximum of 12 characters.

File

A *file* is a set of bytes (on disk) that are treated as a unit. The files of a volume consist of integral numbers of 512 byte sectors and must be completely contained on one disk (up to 1 gigabyte).

You can create a file with the CreateFile operation and delete it with the DeleteFile operation. Once you create a file, you can access it with the OpenFile operation and close it with the CloseFile operation.

The ChangeFileLength operation changes the length of an open file.

The RenameFile operation renames an existing file.

The following factors can influence access to a file:

- The file protection level
- The file password (optional)
- The directory password (optional)

As described in “File Protection,” protection levels and passwords can work together to define file access.

A file name is a string of characters. It can have a maximum of 50 characters.

Password

Four types of password protection are available:

- Volume
- Directory
- File
- Device

A *volume password* protects a volume. A *directory password* protects a directory on a volume. A *file password* protects a file in a directory on a volume. A *device password* is used with operations that work directly with the disk.

You can specify a volume password at the time you initialize the volume using the **Format Disk** command. Use the **CreateDir** operation to specify a directory password. You can specify a file password using the **SetFileStatus** operation.

Volume, directory, and file passwords can consist of all alphanumeric characters, plus the period (.) and the hyphen (-). A volume, directory, or file password can have a maximum of 12 characters.

You can access a file if you know its volume, directory, or file password. Knowing a volume password allows you to access all of the directories and files of that volume. Knowing a directory or file password permits access that is dependent on the file protection level specified for each file.

The **OpenFile** operation accepts a single password. This password is compared first against the volume password, then against the directory password, and last against the file password (if one was specified). You are granted access to open the file if any of these comparisons matches provided the file protection level permits access.

The CreateFile operation accepts a single password that authorizes you to create a file in the specified directory. It is not a password to be assigned to the file being created. This password is compared first against the volume password and then against the directory password. You are granted access to create the file if either of these comparisons matches. (The SetFileStatus operation assigns a password to the file being created. The CreateDir operation assigns a password to the directory being created.)

You can specify a default password using the SetPath operation. The default password is used whenever an explicit password is not specified to an operation. The default password, like an explicit one, is compared to the volume, directory, and file passwords.

Valid passwords are required for some Executive commands, such as Backup Volume, Format Disk, and the User File Editor. If you fail to supply the password or supply an incorrect one, status code 219 ("Access denied") is returned.

The protection provided by each of the four password types is discussed in "Protection by Password."

Directory and File Specifications

You refer to a directory by a directory specification. A *directory specification* has the form

{Node}[VolName]Dirname

You refer to a file by a file specification. A *full file specification* has the form

{Node}[VolName]<DirName>FileName^Password

The distinction between uppercase and lowercase in directory and file specifications is not significant in matching directory and/or file names during directory search; the distinction is, however, preserved by the file management system to make the directory and file specifications easier to read.

It is recommended that node names, volume names, and directory names consist only of alphanumeric characters, the period (.) and the hyphen (-). It is recommended that file names consist of alphanumeric characters, the period (.), the hyphen (-), and the right angle bracket (>).

Abbreviated Specifications

If you previously established a default specification, you can refer to a file or directory by an abbreviated specification.

The SetPath operation establishes a default node, a default volume, a default directory, and a default password. The SetPrefix operation establishes a default file prefix. SetPath and SetPrefix establish defaults for the user number of the caller.

If a program has issued the SetPath operation with the default volume name of [ServerVol] and the default directory name of <Susan>, you can access the files

[ServerVol]<Susan>Todays>work
[ServerVol]<Susan>Yesterdays>work

as either

<Susan>Todays>work <Susan>Yesterdays>work

if just the volume name is omitted, or as

Todays>work Yesterdays>work

if the default volume name and default directory name are omitted; <DirName> cannot be omitted unless [VolName] is also omitted.

If a program has issued the SetPrefix operation with the default file prefix of Todays>, in addition to the default volume name and directory name established by the SetPath operation above, you can access the files

[ServerVol]<Susan>Todays>work
[ServerVol]<Susan>Yesterdays>work

as

work

and

<Susan>Yesterdays>work

You could no longer specify the file in the last example above as

Yesterdays>work

because the file you accessed would be

[ServerVol]<Susan>Todays>Yesterdays>work

which is not the same file.

Automatic Volume Recognition

The operating system automatically recognizes a volume that you placed online (mounted). For example, when you insert a floppy disk into a disk drive, the operating system reads the disk to determine whether it contains a volume and, if it does, that no other volume of the same name is already online. After this validation by the operating system, the volume responds to your requests if they contain the appropriate specifications and passwords.

When a volume is placed online, the operating system reads the volume home block into memory. The volume home block remains there as long as the volume remains online.

If you open a floppy drive door, any open files on the disk in that drive are automatically put into a special dismounted state. You can close such files in the usual manner, but if you attempt to perform other operations on them, status code 235 ("Wrong volume mounted") is returned.

File Protection

The operating system offers a file-oriented security system.

Passwords control access to a specific device, volume, directory, or file. *Protection levels* assigned to each file define the type of access allowed. (For details, see "Protection by Protection Level.")

Using passwords and protection levels together, you can define a file security system to meet your specific needs. Optionally, you can use *volume encryption* to ensure security of passwords of all directories and files created on that volume. (For details, see "Protection by Volume Encryption.")

Protection by Password

The four password types are volume, directory, file, and device. The type of protection provided by each password is described below.

Volume Password

You can access any file, regardless of password or protection level, with the volume password. In the absence of a volume password, the system is not protected. The volume password overrides directory or file passwords. If a volume password exists, it is required for opening the volume as a device.

For example, if you sign on with the volume password, or enter it with the Executive **Path** command, the operating system gives you access to all files on that volume, whether they are password-protected or not, without additional directory or file passwords.

***Note:** You must have a volume password for directory or file passwords to take effect.*

You assign a volume password when you create the volume using the **Format Disk** command. You can change the password using the **Change Volume Name** command.

Directory Password

You can use a directory password to restrict file creation or file renaming within a directory. If a directory password exists, you must specify it or the volume password to create or rename any files within the directory. A directory or volume password is required to remove a directory. You can also use a directory password to access a file, unless a protection level that ignores directory passwords has been assigned to the file. (For details, see "Protection by Protection Level.")

You can establish a directory password with the **CreateDir** or **SetDirStatus** operations.

Use the Executive **Set Directory Protection** command to change or remove a directory password.

File Password

You can use a file password to restrict access to a specific file. If there is a directory password, it becomes the file password when the file is created. In the absense of a directory password, no password is required to open the file. To explicitly assign a file password, the **SetFileStatus** operation can be used. Access to the file depends on the file protection level. By default, a file created with the **CreateFile** operation inherits the password and protection level assigned to the directory in which it is created.

To add a password to a previously unprotected file, or to change a file password, use the Executive **Set Protection** command.

File passwords are most often used to allow certain files in a directory to be read, without allowing access to the other files.

Device Password

You use a device password for operations that work directly with the disk, such as the **Format Disk** or **Backup Volume** commands. The operating system assigns these passwords at system build. For the hard disk, the password is the same as the device name (for example, D0 or D1). For floppy disks, the default is no password.

Using a Password For Access

If you did not assign a volume password to the volume when you initialized it, you can sign on to the system without supplying a password and have full access to all files.

If a volume password was assigned, you can enter a volume, directory, or file password when you sign on. The **SignOn** password is used for access, which is restricted accordingly.

You can also use the Executive **Path** command to enter a password. Thus, if you signed on with a directory password and wish to access files in a different directory, you can supply the necessary password by using the **Path** command. Also, some Executive commands include parameter prompts for a password.

You can also enter a password as a part of a device, volume, directory, or file name. The password consists of the characters between a caret (^) and the end of the parameter or subparameter name, for example:

FileName^Password

Protection by Protection Level

The operating system uses a file protection level to control which types of passwords you are required to supply, if any, to open a specific file in read, peek, or modify mode.

A protection level is assigned only to files. A directory has a default protection level. The default, however, is used to assign a protection level to each file at the time that the file is created.

How Protection Levels Work

Nine protection levels are available. Table 12-1 shows the name, number, and type of access allowed for each protection level. Note that protection level numbers are not hierarchical. Because the password requirements for opening a file in peek and read mode are equivalent, read mode is used to mean either of these modes in the following discussion.

As an example of how protection levels work, the file specified by

[Sys]<MyDir>Foo

is assigned a protection level number of 23 (Nondirectory Modify Password). The *Foo* file, *<MyDir>*, and *[Sys]* are assigned passwords.

You can open the *Foo* file in Read mode without providing a password. You cannot, however, open the *Foo* file in modify mode by providing the password for *<MyDir>*. (Note in Table 12-1 that you must supply either the volume password or the file password to gain access to the *Foo* file in modify mode.)

Table 12-1. Protection Levels

Protection Level	Level Number	Password Required (Read or Peek Mode)*	Password Required (Modify Mode)*
Unprotected	15	None	None
Modify Protected	5	None	Directory
Nondirectory Modify Password	23	None	File
Modify Password	7	None	Directory or file
Access Protected	0	Directory	Directory
Read Password	1	Directory or file	Directory
Nondirectory Access Password	19	Directory or file	File
Access Password	3	Directory or file	Directory or file
Nondirectory Password	51	File	File

*You can access any file with the volume password regardless of password or protection level.

The default file protection level does not affect the passwords and protection of the directory in any way. It is used only as a default level for files created within the directory. If, for example, a directory has a password and is assigned the lowest level of protection (15, unprotected), it is not totally unprotected since you are required to provide a directory or volume password to create or rename files within that directory. When created, files within that directory are assigned a protection level of 15 (unprotected). You can change the protection level with the **Executive Set Directory Protection** command.

How The Operating System Validates Protection Levels

The operating system validates that a file can be opened in read or modify mode. To do this, the operating system first checks if a volume password was provided to open the file. If a volume password was provided, it is compared with the assigned volume password, if any. A match grants access to the file with no further validation.

If, however, a volume password was not provided, the operating system checks the protection level number against a bit pattern. The bit pattern is described in Table 12-2.

Bit numbers 0 through 7 designate the file protection level, as shown in the table.

The operating system checks the meanings of these bits against the password information (directory password, file password, or none) supplied to open the file. If any of the bit checks is valid, the file can be opened. Otherwise, status code 219 ("Access denied") is returned.

Table 12-2 Bit Number Designations for Protection Level

Bit	Meaning
Bit 0:	If the value is 1 and if there is a file password, it is valid for opening in read mode (mr).
Bit 1:	If the value is 1 and if there is a file password, it is valid for opening in modify mode (mm).
Bit 2:	If the value is 1, no password is required for read mode (mr).
Bit 3:	If the value is 1, no password is required for modify mode (mm).
Bit 4:	If the value is 1, a directory password is not valid for modify mode (mm).
Bit 5:	If the value is 1, a directory password is not valid for read mode (mr).
Bit 6:	Reserved for internal use.
Bit 7:	Reserved for internal use.

As an example, the file specified by

[Sys]<MyDir>Foo

is assigned protection level number 15.

15 (0Fh) in binary form is

Bit numbers: 7 6 5 4 3 2 1 0

Bits set: 0 0 0 0 1 1 1 1

Bit numbers 2 and 3 are set. This means the *Foo* file can be opened in read or modify mode without a password. Note that this agrees with protection level 15 (unprotected). (See Table 12-1 in “How Protection Levels Work.”)

As another example, if the *Foo* file above is assigned protection level number 51 (33h); in binary form, this is

Bit numbers: 7 6 5 4 3 2 1 0

Bits set: 0 0 1 1 0 0 1 1

In this case, bits 2 and 3 are 0. As a result, a directory or a file password is required to open the file in read or modify mode.

The operating system then checks for a password supplied to open *Foo*. It matches this password with the one assigned.

Because bits 4 and 5 are set, a matching directory password is not valid. Bits 0 and 1 also are set, however, indicating that a matching file password is valid for opening the file.

Note that the bit interpretations agree with protection level 51 (nondirectory password). (See Table 12-1 in “How Protection Levels Work.”)

(For details on common system protection applications, see the *CTOS System Administration Guide*.)

Protection by Volume Encryption

You can use the **Format Disk** command option to encrypt the passwords of all files and directories created on a volume. (For details on the **Format Disk** command, see the *CTOS Executive Reference Manual*.)

An encrypted password has the following characteristics:

- The password is at most 12 bytes (that is, 12 characters long).
- The high-order bit is set in the byte for the rightmost character.

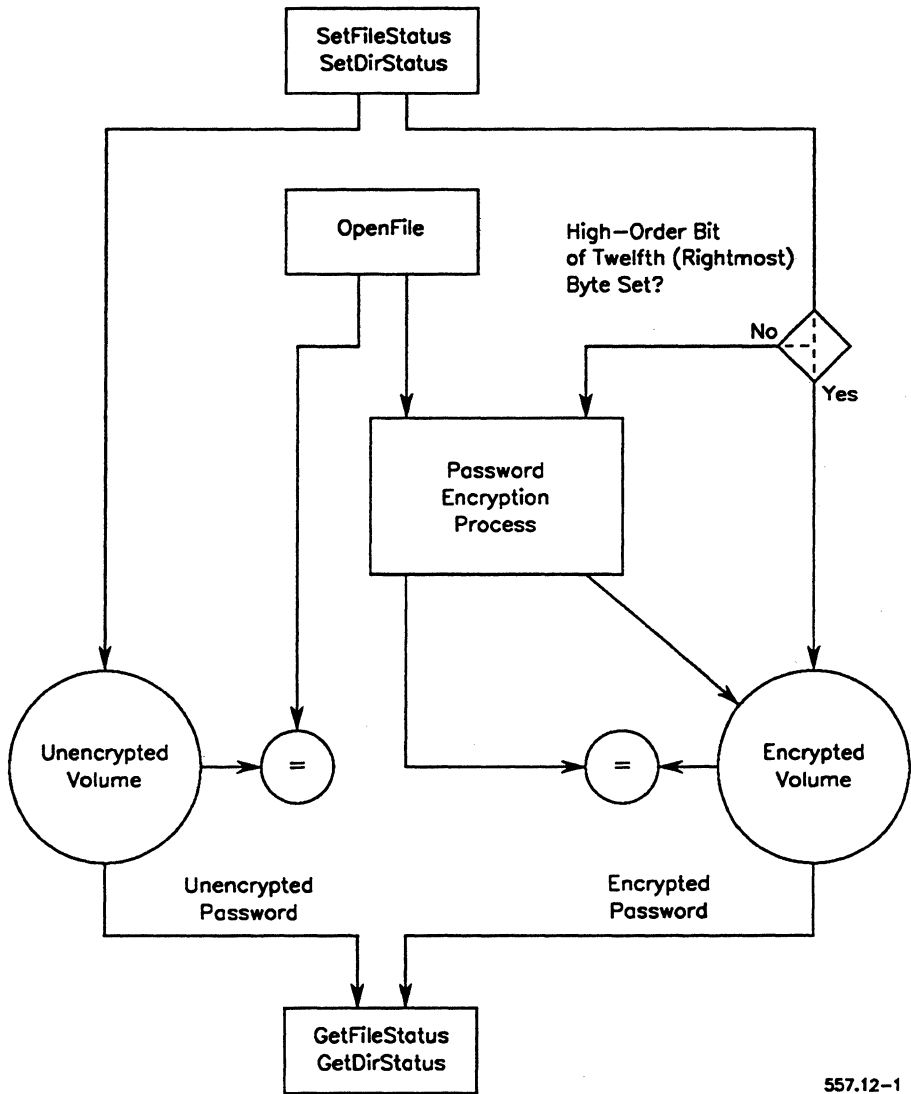
Figure 12-1 compares the effects of volume encryption on operations that require passwords.

All passwords provided to the **OpenFile** operation are encrypted for an encrypted volume.

The **GetFileStatus** and **GetDirStatus** operations return encrypted file and directory passwords, respectively, for an encrypted volume.

Note that pressing **CODE** in combination with another key results in setting the high-order bit of a byte. Using this key combination for the rightmost character of a 12 character password string is not recommended. This is because the **SetFileStatus** and **SetDirStatus** operations interpret such a password as encrypted for an encrypted volume. Access using this password would be denied in a future **OpenFile** operation. (See Figure 12-1.)

Figure 12-1. Effects of Volume Encryption



557.12-1

Creating and Accessing a File

You can create and access a file on a disk device using the program interface levels listed below:

- Structured file access methods
- Byte streams (Sequential Access Method)
- File management operations

Structured File Access Methods

The structured file access methods provide access to data files that are structured in specific ways. A section is dedicated to each of these methods in this manual. (For details, see the section entitled “Structured File Access Methods.”)

Byte Streams

You can create and access disk files by using the Sequential Access Method (disk byte streams).

When you use disk byte streams, you are using the file management operations indirectly. The byte stream routines call the appropriate file management operations for you. You can write as many bytes as you want (provided you do not run out of disk space). When you close your file, the byte stream makes the appropriate calls to close the file.

Most programs use the byte stream interface level because it is a relatively easy and flexible way to create and to access files. (For details, see the section entitled “Sequential Access Method.”)

File Management Operations

At the very lowest interface level (closest to the hardware), you can use the file management operations described in “File Management Operations.” At this level, you have the greatest degree of control over the file you create. You can also use the Request and Wait primitives and build your own request block based on the request blocks for these operations.

The file management operations provide random access to 512 byte sectors of a file. (512 bytes is the size of a physical disk sector.) The operations allow you to read and write multiple sectors, starting with a particular sector of a file. Device independence is provided by masking the device characteristics of the disk on which the file is located. (Use of the file management operations is discussed in “Creating, Using, and Deleting Files.”)

Logical File Address

A *logical file address* (lfa) is a 32 bit unsigned integer that your program uses to locate a position within a file. It specifies a byte position; that is, it is the number (the offset) that would be assigned to a byte in a file if all the bytes were numbered consecutively starting with 0.

You use the lfa in file management operations (such as Read or Write) to locate a particular sector of a file. The lfa must be on a sector boundary. Therefore, you must supply an lfa (in bytes) to a Read or a Write operation that is a multiple of 512. For example, to locate the third sector in a file, you would supply an lfa of 1024.

If you are using byte streams, however, you are not required to provide an lfa that is a 512 byte multiple.

The 2 high-order bits of the lfa are reserved as special indicators. Bit 31 is set to override normal system checks and is used to attempt access to defective disks. Bit 30 is set to suppress retry of input or output to recover from errors. For example, a program logging high-speed, digitized wave forms that could accept badly written data but not the time required for retry, would specify an lfa of 40000400h to specify the third sector of a file with error retry suppressed. The returned status code reports errors in the normal way even when the special indicators are set.

File Handle

A *file handle* (fh) is a 16 bit integer that, in combination with a user number, uniquely identifies an open file. File handles are returned by the `OpenFile` operation and are used to refer to a file in subsequent operations such as `Read`, `Write`, and `DeleteFile`. (To use a file handle on behalf of another user number, the caller must use an `Alt` request. For details, see “Using the Request Procedural Interface,” in the section entitled “Interprocess Communication.”)

A file handle can be long-lived or short-lived. You can use the `OpenFileLL` or `SetFhLongevity` operation to set a file handle long-lived. Only a short-lived (normal) file handle is closed by a `CloseAllFiles` operation or automatically when an application program terminates. A long-lived, as well as a short-lived, file handle is closed by an explicit `CloseFile` operation or by the `CloseAllFilesLL` operation.

From Creating to Deleting a File

Your application can perform the following procedures on a file using the file management operations:

1. Creating a file.
2. Opening a file.
3. Writing data to a file and subsequently reading the data.
4. Closing a file.
5. Deleting a file.

Each of these procedures is described below. A comparable description is given for performing each procedure using disk byte streams.

Creating a File

What You Do to Create a File

To create a file using the file management operations, you need to call `CreateFile`. You can specify the length of your file as a multiple of 512 bytes, or you can specify 0 bytes. If you specify 0 bytes, you must make a subsequent call to the `ChangeFileLength` operation to specify the file length.

The `CreateFile` and the `ChangeFileLength` operations are the only operations that allocate disk sectors for a file. `ChangeFileLength` can allocate or deallocate sectors. The operating system uses the byte value you specified to determine the number of 512 byte sectors to allocate for your file.

When you use the byte streams interface, the byte stream automatically calls the `CreateFile` operation. A byte stream always creates a 30 sector file, expanding the file in 30 sector increments, as required. When the file is closed, file size is contracted to the end of the sector containing the lfa of the last byte written (end-of-file pointer).

What the Operating System Does to Create a File

The operating system performs the following steps when you call the operations `CreateFile` and `ChangeFileLength`. The operating system

1. Verifies that a volume of the requested name is already online. (The volume home block is brought into memory when a volume is placed online. For details, see “Volume Control Structures.”)
2. Verifies that a directory of the requested name is on that volume. (The most recently used directory information is retained in memory.)
3. Verifies that a file of the requested name does not exist in that directory. (The most recently used file information is retained in memory.)
4. Allocates a file header block and assigns the requested number of disk sectors by consulting the allocation bit map. (The allocation bit map controls the assignment of disk sectors. For details, see “Volume Control Structures.”)
5. Inserts an entry for the file in the requested directory.

Opening a File

What You Do to Open a File

To open a file using the file management operations, you call the `OpenFile` or the `OpenFileLL` operation. In either case, you supply the file specification, the password (if required), and the file mode. A file handle is returned to your program that you can use in future requests (such as `Write` or `Read`) to the opened file.

Note that the byte stream interface opens the file for you when you open the byte stream.

What the Operating System Does to Open a File

When you open a file, the operating system

1. Verifies that a volume of the requested name is already online. (The volume home block is brought into memory when a volume is placed online.)
2. Verifies that a directory of the requested name is on that volume. (The most recently used directory information is retained in memory.)
3. Verifies that a file of the requested name is in that directory. (The most recently used file information is retained in memory.)
4. Allocates a file control block, one or more file area blocks, and a user file block (UFB). (For details on these structures, see "System Data Structures.")
5. Copies the information from the file header block to the file control block and one or more file area blocks.
6. Returns a file handle. The file handle serves to identify this particular file control block.

Reading and Writing a File

Using the File Management Operations

You can select to read from and write to a file in three ways when you use the file management operations. These are

- Using the Read and Write operations. The Read and Write operations are the simplest way of performing I/O, because constructing a request block and issuing the Request and Wait primitives are done automatically. Read and Write do not provide for any overlap between I/O operations and computation.
- Using the ReadAsync, CheckReadAsync, WriteAsync, and CheckWriteAsync operations. The ReadAsync and WriteAsync operations are a more complex way of performing I/O. They allow a program to initiate an I/O transfer and then compute and/or initiate other I/O transfers before checking (with the CheckReadAsync and CheckWriteAsync operations) for the successful completion of the first transfer.
- Constructing a request block and using the Request and Wait (or Check) primitives. This is the most direct method of reading and writing a file. It also requires the most effort on your part. This method allows your program to overlap multiple I/O operations and computation.

(See the section entitled “Interprocess Communication,” for details on the Request, Wait, and Check Kernel primitives.)

When you write to the file, you must specify where in the file your data is to be written. You can write full sectors only. However, you can write to any byte offset in the file that is a multiple of 512 (beginning of a sector).

If you write more data than can be contained within the number of sectors allocated, you must allocate more sectors by calling `ChangeFileLength` and supplying the new file length.

If you write to fewer sectors than you created, you can call `ChangeFileLength` to change the file length to a new shorter length.

Your program, however, may require the unused sectors as temporary space for holding variable amounts of data at different times. In such a case, it would be to your advantage to retain the extra sectors. If you anticipate frequent changes to the file length, you should consider the following:

- Each time you change the sector length of your file, the operating system has to allocate or deallocate sectors and consult its allocation bit map. (For details on the allocation bit map, see “Volume Control Structures.”)
- Frequent changes to the allocation bit map fragment the disk space.

If you plan to use your disk file as input to a program that uses byte streams, you must call the `SetFileStatus` operation to specify the logical file address of the last byte you wrote (end-of-file pointer). `SetFileStatus` is used to set the end-of-file pointer only. To allocate additional sectors, you must use the `ChangeFileLength` operation.

Using Byte Streams

When you write to a file using the byte stream interface, you can write any number of bytes (versus being restricted to multiples of 512). The operating system writes your data sequentially to the disk.

When you append data to the file using byte streams, the data is written where the previous data ended.

Random access using byte streams is not as efficient as it is when using the file management operations. This is because you do not have as much control over the amount of data being read.

Closing a File

Using the File Management Operations

When you have completed the processing of a file, you close it using the operations `CloseFile`, `CloseAllFiles`, or `CloseFilesLL`. The number of 512 byte sectors allocated for the file is not changed.

If, for example, you had written 512 bytes and the file length that you specified to your last `ChangeFileLength` operation was 1024 bytes, the file length will remain 1024 bytes when you close the file.

Using Byte Streams

When you close the byte stream, the end-of-file pointer is set automatically. The byte stream adjusts the number of allocated file sectors to the minimum required to contain your file data.

If, for example, you closed a file containing 612 bytes of data, the byte stream calls `SetFileStatus` to set the end-of-file pointer within the second sector. `ChangeFileLength` then is called to deallocate the unused 28 sectors.

Deleting a File

Using the File Management Operations

When you have no more need for a file, you delete it. If you initially opened the file in modify mode (exclusive access), you can use the `DeleteFile` operation to delete it. `DeleteFile` frees the FHB, deallocates file sectors, and removes the directory entry for the file.

If the file is shared by several user numbers (for example, database and software installation files), you can delete it with *statusCode* 14 (the file removal status byte) of the `SetFileStatus` operation. To delete a file with `SetFileStatus`, your application must have opened the file using a password that permits reading and modifying the file.

Setting *statusCode* 14 removes the file directory entry when you close the file but does not free the FHB or deallocate the file sectors until all the user numbers that have opened the file have closed it as well. To reverse this effect, you may clear *statusCode* 14 at any time before closing the file.

Using Byte Streams

The `DeleteByteStream` operation allows your application to delete a file using byte streams.

Local File System

When the operating system intercepts a request to open a file, it routes the request to the local file system. If the volume is not found, it routes the request to the server.

You can route a file access request explicitly to the server by including the special exclamation point character (!) before the volume specification, as in *[/!Sys]<Sys>Exec.Run*, for example.

Any cluster workstation can access files on disks at the server. However, you cannot access files on a local disk from the server or from other cluster workstations. You must copy a local file to the server if it is to be processed by the server, another workstation in the cluster, or another node.

You must copy a local file to the server before it can be processed by any of the following:

- Spooler (if the Generic Print System is not in use)
- Remote job entry (RJE)
- Indexed Sequential Access Method (ISAM)
- Any system service executing at the server or another cluster workstation

A cluster workstation bootstrapped from its local file system is a self-contained entity that must access the server only for shared files. If a malfunction occurs at the server, the cluster workstation can continue to operate normally provided all file accesses are to local disks.

LfsToMaster

LfsToMaster is a system configuration file option that provides for sharing server run files with cluster workstations. (For details on configuration file options, see the *CTOS System Administration Guide*.)

LfsToMaster results in certain requests for opening a file that fail locally to be retried at the server. The request is retried if all of the following conditions are TRUE:

- The request is an OpenFile, OpenFileLL, or ReopenFile operation opened in read or peek (shared) mode.
- The status code returned is 203 (“No such file”).
- The request originated at a cluster workstation with a local file system.
- The file specification is of the form *[Sys]<Sys>Filename*.

To specify the local file system (and thereby override the default of routing the request to the server), use *[+Sys]<Sys>Filename* as the file string.

Volume Control Structures

A disk volume is formatted to contain volume control structures.

Volume control structures allow the file management system to manage (allocate, deallocate, locate, avoid duplication of) the space on the volume not already allocated to the volume control structures themselves.

Volume control structures are created when the disk is first initialized. Initialization must be performed using the **Format Disk** command. (For details, see the *CTOS Executive Reference Manual*.)

The volume control structures include the

- Volume home block
- File header blocks
- Master file directory
- Directories
- Allocation bit map

The primary and secondary copies of the file header block are located on different cylinders and at different rotational positions and are accessed (except for floppy disks) by different read/write heads. These duplicates ensure that damage to one copy does not cause a data loss. The **Format Disk** command permits suppression of duplicate file header blocks. However, this reduces reliability and is not recommended.

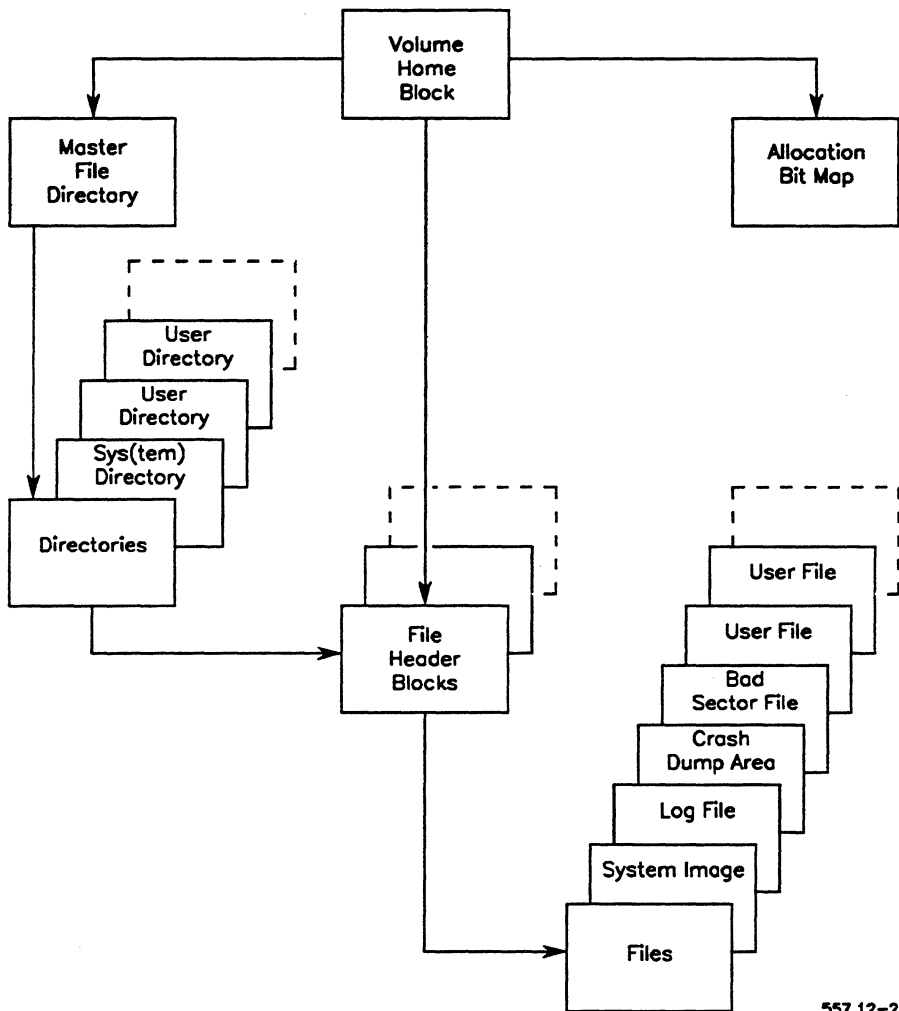
The initial copy, unlike the working copy of the volume home block, is not modified after it is created. The primary and secondary copies of the file header block, however, are always true duplicates.

Volume Home Block

Each volume is assigned a volume home block. The volume home block (VHB) is the root structure (that is, the starting point for the tree structure) of the information on the disk volume and is one sector in size. The VHB contains the volume name and the date it was created. In addition, it contains the memory addresses of the master file directory, the allocation bit map, the file header blocks, the System Image, the log file, the crash dump area, and the bad sector file. The relationship of the VHB to the other volume control structures is shown in Figure 12-2.

(For details on each field in the VHB, see "Volume Home Block" in the section entitled "System Structures," in the *CTOS Procedural Interface Reference Manual*.)

Figure 12-2. Volume Control Structures



557.12-2

Allocation Bit Map and Bad Sector File

The allocation bit map controls the assignment of disk sectors. It has 1 bit for every sector on the disk, and the bit is set if the sector is available. The size of the allocation bit map depends on the size of the volume.

The operating system places an entry for each unusable disk sector in the bad sector file. The bad sector file is one or more sectors in size.

File Header Block

Each file is assigned a file header block (FHB). The FHB contains information about the file such as its name, password, protection level, the date/time it was created, the date/time it was last modified, and the disk address and size of each of its disk extents. The FHB is one sector in size.

(For details on the FHB structure, see “File Header Block” in the section entitled “System Structures,” in the *CTOS Procedural Interface Reference Manual*.)

Disk Extent

A *disk extent* is one or more contiguous disk sectors that compose all or part of a file. The entry for a disk extent in the FHB is 8 bytes: 4 bytes specify its location, and 4 bytes specify its size.

The operating system allocates a file area block (FAB) for each disk extent of an open file.

Extension File Header Block

A FHB can accommodate 32 disk extents. A file that contains more requires *extension file header blocks* (extension FHBs). Extension FHBs are seldom necessary unless you place an unusually heavy burden on the file management system. Your file may require extension FHBs, for example, if you expand the same file many times or fragment the available disk space by deleting and creating files frequently on a nearly full volume you seldomly refresh. (You can refresh a volume by using the **Backup Volume**, **Format Disk**, and **Restore** commands. See the *CTOS Executive Reference Manual* for details on these commands.)

Master File Directory and Directories

Each directory on a volume, including the Sys directory (see below), has an entry in the master file directory (MFD). The entry's position within the MFD is determined by randomization (hashing) techniques. The entry contains the directory's name, password, location, and size.

(For the format of a directory entry in the MFD, see "Directory Entry in Master File Directory Block" in the section entitled "System Structures," in the *CTOS Procedural Interface Reference Manual*.)

Each directory on the volume consists of one or more directory sectors. Randomization (hashing) techniques determine the directory sector in which the file is entered. The file entry contains the file's name and a pointer to the FHB.

(For the format of a file entry, see "File Entry in a Directory Block" in the section entitled "System Structures" in the *CTOS Procedural Interface Reference Manual*.)

The MFD and the directories provide fast access to the file header block of a specific file. They do not, however, contain any information about the file that is not also contained in its FHB. (The most recently used file and directory information is retained in memory.)

System Directory

The Sys directory is different from other directories in two ways. First, when a volume is initialized, its MFD contains only one entry, which is for the Sys directory. (You can create other directories by using the CreateDir operation.) Second, the Sys directory contains entries for all system files. You must not delete, rename, or overwrite these files.

These file entries are always present in the Sys directory of each volume:

- the MFD (*Mfd.sys*)
- the FHB (*FileHeaders.sys*)

Depending on how you format the disk, other entries may be present (for example, *CrashDump.sys*, *Log.sys*, and *SysImage.sys*).

System Data Structures

System data structures are data areas contained within the operating system and are necessary for its operation. They are often configuration-dependent. The six system data structures related to the file management system are listed below:

- User control block (UCB)
- Device control block (DCB)
- File control block (FCB)
- File area block (FAB)
- I/O block (IOB)
- Volume home block (VHB)

The UCB and the DCB are user-accessible and are described below.

User Control Block

A *user number* is associated with the resources allocated to an application partition.

Each user number is assigned a UCB. The UCB contains the default node, default volume, default directory, default password, and default file prefix set by the last SetPath and SetPrefix operations.

(For the format of the UCB, see “User Control Block” in the section entitled “System Structures,” in the *CTOS Procedural Interface Reference Manual*.)

Device Control Block

Each physical device is assigned a DCB. The DCB contains information about the device. Information contained in the DCB is obtained in the following ways:

- Generation at system build
- Derivation from the volume home block
- Dynamic calculation by the operating system

For a disk, the information may include the number of disk tracks, the number of sectors per track, and so forth. The DCB contains the memory address of a chain of I/O blocks.

(For the format of the DCB, see “Device Control Block” in the section entitled “System Structures,” in the *CTOS Prodedural Interface Reference Manual*.)

Building and Parsing File Specifications

The standard operating system library contains operations for parsing and building file specifications. In addition, there are utility procedures for performing related functions.

It is recommended that you use these operations in all applications you write that manipulate file specifications. For compatibility with internationalized standards, all programs are required to use these procedures.

Terms

When referring to the parts of a file specification, the discussion of building and parsing employs the following new terms:

Token indicates a node, volume, directory, or file name string, or a password string. A token excludes any brackets surrounding the string. As an example, in the file specification below, the volume token is *Sys* and the directory token, *MyDir*:

[Sys]<MyDir>

Each token represents a *level* within the hierarchical organization of disk file data. From top to bottom, these levels are node, volume, directory, and file name.

Using The File Building and Parsing Operations

The file parsing operations search through a file specification for each level requested by the caller. After parsing each level and removing the brackets surrounding tokens, each requested token and its size (in bytes) are returned at memory addresses provided by the caller. The building operations build a file specification from tokens selected and passed by the caller.

Say, for example, your application accepts user input to change the default path. Furthermore, your application allows the user to enter the path specification on a single command line. To set the new default path, the *SetPath* operation requires that individual tokens for each specification level be passed to it. (For details on the parameters to *SetPath*, see the *CTOS Procedural Interface Reference Manual*.) In this situation, your application would need to validate each level the user types as well as to extract the tokens so that they can be passed to the *SetPath* operation. Historically, there was no uniform set of rules an application could use to do this. With the file parsing operations, applications can now parse specifications and perform the necessary validation in a consistent manner.

Let's assume the current path is *[MyVol]<OldDir>*. To change the path, the user types the following specification on the command line:

[MyVol]<NewDir>

To verify this specification and parse the individual tokens *MyVol* and *NewDir*, your application can call a parsing operation such as `ParseFileSpec`. Then, before setting the new path, your application can check the validity of the new path (verify that there is a directory called *NewDir* on *MyVol*). To accomplish this, your application can first call a file building operation such as `BuildFileSpec` to build a specification with the volume and directory tokens, *MyVol* and *NewDir*. Following the call to `BuildFileSpec`, `GetDirStatus` can be called with the specification returned to validate the path.

Layers of Access

The parsing and building operations actually consist of two layers of operations. In the top layer, `ParseFileSpec` and `BuildFileSpec` perform all the functions necessary for parsing and building file specifications, respectively. Through parameters specified to either of these operations, your application can select the file specification levels to be parsed or built.

`ParseFileSpec` and `BuildFileSpec`, in turn, call a lower layer of parsing and building operations. Each lower-layer operation is called to handle a single level of building or parsing. For example, `ParseFileSpec` calls `ParseSpecForNode` to obtain the node token and `ParseSpecForVol` to obtain the volume token. `BuildFileSpec` calls the corresponding building operations, `BuildSpecFromNode` and `BuildSpecFromVol`, to obtain the node and volume tokens, respectively, to build a file specification.

`ParseFileSpec` returns each token at a unique address for use in operations (such as `SetPath`) that require the addresses of individual tokens be passed to them. `BuildFileSpec`, on the other hand, returns the full file specification at the address specified. `BuildFileSpec` builds the specification by appending each token in hierarchical order and inserting the appropriate brackets.

If your application only requires building or parsing a subset of file specification levels, the lower-layer operations can be called directly. If you use these lower-level operations in your application, however, your program may be required to perform extra work normally accomplished by setting flags in `ParseFileSpec` or `BuildFileSpec`.

For example, you can set the flag *fDefaultPath* in *ParseFileSpec* or *BuildFileSpec* to specify that the default path token be returned in the event that a level being parsed is not present in the file specification provided to these operations. If your application calls a lower-layer operation, on the other hand, your application is responsible for calling the *GetUcb* operation to obtain the memory addresses of the default path tokens before calling the lower-layer operation. (For details on *GetUcb* see the *CTOS Procedural Interface Reference Manual*.)

ParseFileSpec or *BuildFileSpec* also provide the flag *fCanonical*. This flag can be set to return the canonical form of a node or volume token, such as *Win* for the volume token *Sys*. To obtain the canonical node or volume token using the lower layer of operations requires that your application make two additional calls to the utility operations, *RemoveFfsBrackets* and *GetCanonicalNodeAndVolume*.

There is, in addition, other information you can pass to *ParseFileSpec* or *BuildFileSpec* so that these operations can handle parsing and building in a uniform way. For example, you can specify which levels are assumed to be present by passing an appropriate value for the parameter *bSpecType*. (For details, see guideline 6 in "Validating File Specifications.")

Validating File Specifications

When an application calls a parsing or building operation, the application cannot always assume that the input file specification passed is formatted correctly. For example, a file specification entered by the user at the keyboard may contain extraneous brackets, or levels could be transposed. Some validation is provided by the file system. As a further check, the parsing and building operations employ a common set of guidelines for validating input. These guidelines are described below.

Note: *Because each parsing and building operation only verifies errors in that portion of a file specification passed to it, validation performed by these operations may not detect all errors. (See guideline 4, below.)*

1. Because the delimiting brackets within a file specification are unique, it is relatively easy for the parsing or building operations to identify the different file specification levels. Each level is recognized by the leading and trailing brackets and includes the characters enclosed within the brackets. In a full file specification such as *{Local}[Sys]<Dir1>Foo^1234*, the characters at each level are identified as shown below:

Level	Characters at this level
Node	<i>{Local}</i>
Volume	<i>[Sys]</i>
Directory	<i><Dir1></i>
File name	<i>Foo</i>
Password	<i>^1234</i>

2. A token is defined as the characters between the leading bracket and trailing brackets of the level. In the preceding full file specification, for example, the tokens are as shown below:

Level	Token at this level
Node	<i>Local</i>
Volume	<i>Sys</i>
Directory	<i>Dir1</i>
File name	<i>Foo</i>
Password	<i>1234</i>

3. A file specification is parsed and built based on the following specification format:

{Local}[Sys]<Dir1>Foo^1234

The password, however, can follow any level and is always considered the last level in the file specification, for example,

{Local}[Sys]^1234

4. Validation occurs only at the level being handled by a parsing or building operation. If, for example, the caller requests that the node and volume levels be parsed, an error is not detected in the directory or file levels. In the following file specification, a syntax error is returned only if the caller requests that the directory level be parsed:

[Sys]<<Dir1>>Foo

For a description of the syntax errors, see “Syntax Errors.”

5. Brackets are illegal characters in a file token. (An exception to this rule is the right angle bracket (>), which is commonly used as a prefix delimiter in a file name.)
6. Incomplete file specifications are easily parsed, provided the unique brackets for each level are present. If, however, your application accepts incomplete file specifications, you must provide information to the parsing and building operations on the levels assumed to be present by passing an appropriate value for the parameter *bSpecType*. As an example, the **Create Directory** command allows a specification of the form [d1]Dir1. By specifying a value of 2 for *bSpecType*, a specification is accepted that only contains a volume and directory name or simply a directory name. (For details on all the values of *bSpecType*, see the description of ParseFileSpec in the *CTOS Prodedural Interface Reference Manual*.)

Syntax Errors

All the parsing and building operations return a word value to the caller rather than an error code. The word returned indicates the status of the validation performed. If no errors are detected, a value of 0 is returned.

The first 6 bits in the low-order byte of the word correspond to errors that are detected at particular levels of the specification. If bit 0 is set, for example, an error was detected at the node level. More than one bit can be set in cases where there are errors at multiple levels, such as may occur when ParseFileSpec or BuildSpec are called to parse or build several levels at the same time. Table 12-3 describes the meanings of the bits when set.

Table 12-3. Syntax Errors

Bit Set	Error
0	Bad Node Level
1	Bad Volume Level
2	Bad Directory Level
3	Bad File Level
4	Bad Password Level
5	File Specification Too Long

Wild Card Operations

A *wild card* is a special character in a file specification. It instructs the Executive program to search for file specifications that match all characters given in the file specification except the wild card character(s).

The Executive recognizes the asterisk (*) and the question mark (?) as wild card characters. (For details on wild card characters, see the *CTOS Executive Reference Manual*.)

The following wild card operations can be used with files:

- WildCardInit
- WildCardNext
- WildCardClose

You can use these operations to build a list of files that match a wild card specification.

To do this, your program can call WildCardInit with a wild card specification. Then it can work in a loop, calling WildCardNext. Each time WildCardNext returns to your program, it returns to the next file name that matches the wild card specification. WildCardClose should always be called to release the buffer used with WildCardInit and WildCardNext.

The Executive, for example, uses these operations to expand wild cards that the Executive user types into a form.

\$ Directory

The `<$>` *directory* is a disk directory in which programs can create temporary files. A `<$>` directory is required by all application programs and is needed for the maximum number of users connected to the server.

When a request with the directory name of `<$>` is given as part of a file specification, the operating system expands the directory name to the form

`<$000>nnnnn`

where *nnnnn* is the user number associated with the application partition. This expansion occurs only if the directory name is `<$>`.

If, for example, user number 3 requests access to the file `Foo` on the `[Sys]` volume using the directory name `<$>`, the file specification is expanded as follows:

`[Sys]<$>Foo` to `[Sys]<$000>00003>Foo`

Since the user number(s) of a cluster workstation are reassigned whenever the system is bootstrapped, you should not use the `<$>` directory for permanent files.

File Management Operations

The file management operations are described below. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Basic

OpenFile

Opens an already existing file, and returns a file handle.

Read

Transfers an integral number of 512 byte sectors from disk to memory.

Write

Transfers an integral number of 512 byte sectors from memory to disk.

CloseFile

Closes an open file.

CloseAllFiles

Closes all files that are currently open for the user, except those marked long-lived.

Basic Utility Operations

CreateFile

Creates a file of the specified name in the specified directory on the specified volume.

DeleteFile

Deletes an open file.

RenameFile

Changes the file name and/or the directory name of an existing file. A file can be renamed to another directory on the same volume.

WildcardClose

Closes a wild card session by releasing the buffer used by WildCardInit and WildCardNext.

WildcardInit

Establishes a wild card file specification to be used by subsequent calls to the related WildCardNext operation.

WildcardNext

Returns the next file name that matches a wild card file specification supplied previously by a call to WildCardInit.

AtFileInit

Opens and initializes a file containing character strings separated by white space (spaces, tabs, carriage returns). Each string in the file then is processed by subsequent calls to AtFileNext.

AtFileNext

Returns the next string contained in the file opened previously by a call to AtFileInit.

File Attributes

ChangeFileLength

Expands or contracts an open file to a new length.

GetFileInfoByName

Provides the capability to determine the canonical file specification and device name of a file without opening it.

GetFileStatus

Copies the requested status information to the specified area.

SetFileStatus

Copies the specified status information from the specified memory area to the FHB.

Default Path

ClearPath

Clears the defaults established by the SetPath and SetPrefix operations.

SetPath

Establishes a default volume, a default directory, and a default password.

SetPrefix

Establishes a default file prefix that begins the file name part of a file specification if that file specification does not have an explicit volume or directory name.

SetNode

Allows the specification of a node name to be used as part of the default path whenever a file specification is given that does not contain a node or volume name.

GetUcb

Copies the UCB for the current user number to the specified area.

Directories

CreateDir

Creates a directory of the specified name on the specified volume.

DeleteDir

Deletes an empty directory.

ReadDirSector

Reads a 512 byte sector of the specified directory.

GetDirStatus

Determines information about a directory.

SetDirStatus

Changes a directory password or default file protection level.

GetDirInfo

Copies the requested directory entry from the master file directory to the specified memory area.

Long-Lived Files

OpenFileLL

Opens an already existing file and returns a file handle marked long-lived.

SetFhLongevity

Sets a file handle as long-lived or short-lived.

GetFhLongevity

Copies the requested information on the longevity of the file handle to the specified area.

CloseAllFilesLL

Closes all files that are currently open for the user, including those marked long-lived.

File Handle Operations

ChangeOpenMode

Changes the access mode of a file that is already open.

RemakeFh

When given an existing file handle, creates a new file handle to be associated with the user number of the process issuing this request.

ReopenFile

Is similar to *OpenFile* with the following exception: if a file handle already exists for the issuing user number's file, that handle rather than a new one is returned.

Parsing Specifications

ParseFileSpec

Searches a file specification and returns each token (that is, node, volume, directory, or file name string or password string) and the token size for each level specified.

ParseSpecForNode

Searches a file specification to find the node level, removes the brackets at this level, and stores the node token and its size at memory addresses provided by the caller.

ParseSpecForVol

Searches a file specification to find the volume level, removes the brackets at this level, and stores the volume token and its size at memory addresses provided by the caller.

ParseSpecForDir

Searches a file specification to find the directory level, removes the brackets at this level, and stores the directory token and its size at memory addresses provided by the caller.

ParseSpecForFile

Searches a file specification to find the file name level and stores the file name token and its size at memory addresses provided by the caller.

ParseSpecForPassword

Searches a file specification to find the password level, removes the caret(^), and stores the password token and its size at memory addresses provided by the caller.

Building Specifications

BuildFileSpec

Builds a file specification from the tokens (node, volume, directory, file name, and password strings) passed to it by the caller. The file specification with the appropriate brackets inserted and its size are returned at the specified memory addresses.

BuildSpecFromNode

Adds the node to the output file specification returned to the caller.

BuildSpecFromVol

Adds the volume to the output file specification returned to the caller.

BuildSpecFromDir

Adds the directory to the output file specification returned to the caller.

BuildSpecFromFile

Adds a file name to the output file specification returned to the caller.

BuildSpecFromPassword

Adds a password to the output file specification returned to the caller.

BuildFullSpecFromPartial

Returns the full file specification based on the partial specification passed by the caller.

Parsing and Building Specifications

GetCanonicalNodeAndVol

Converts a node and/or volume token (that is, node and/or volume name string) into its canonical form.

RemoveFfsBrackets

Removes the brackets surrounding a token at the specified level (node, volume, directory, or file name) in the file specification.

Asynchronous File I/O

ReadAsync

Initiates the transfer of an integral number of 512 byte sectors from disk to memory. The operation **CheckReadAsync** must be called to check the completion status of the transfer.

WriteAsync

Initiates the transfer of an integral number of 512 byte sectors from memory to disk. The operation **CheckWriteAsync** must be called to check the completion status of the transfer.

CheckReadAsync

Waits for input completion, checks the status code, and obtains the byte count of data read after a **ReadAsync** operation.

CheckWriteAsync

Waits for output completion, checks the status code, and obtains the byte count of data written after a **WriteAsync** operation.

Volume Data Structures

GetVhb

Copies the VHB of the specified device to the specified memory area.

Section 13

Disk Management

What is Disk Management?

Disk management operations provide device-level access to disk devices, in contrast to the file-level access provided by file management operations. Access to a disk device at such a level may be necessary to read a floppy disk written on a foreign operating system or to format an uninitialized disk.

Device-level access is provided to the following media:

- Single or dual sided, 3 1/2 inch and 5 1/4 inch floppy disks written in double or high density
- Most varieties of hard disks
- Memory disks
- SCSI disks

The sector size and density of a floppy disk, if other than 512 byte, double-density, must be specified with the SetDevParams operation. (For a complete description of SetDevParams, see the *CTOS Procedural Interface Reference Manual*.)

Accessing a Disk Device

A disk device can be accessed using an operation that opens a file or device (for example, OpenFile) and providing a device or volume specification. All of the following operations accept a resource handle returned by such an operation:

- CheckReadAsync
- CheckWriteAsync
- CloseFile
- GetCtosDiskPartition

Read
ReadAsync
Write
WriteAsync

(For details on resource handles, see “Routing by Handle,” in the section entitled “Interprocess Communication.”)

Device-level access to disks bypasses the concurrency control of the file management system. Thus extreme care is required if device-level access is used in a cluster configuration.

Device Specification and Password

A disk device is a physical hardware entity or, in the case of memory disks, a software memory area. Access to a device requires presentation of a device specification and a password. A device specification can take either of two forms, depending on whether the medium of the disk device contains a valid file system.

If a volume contains a valid file system, the device specification has the form

{Node}[VolName] or {Node}[DevName]

In this case, the volume password must be specified. Volume passwords are described in “Protection by Password,” in the section entitled “File Management.”)

If, however, the medium does not contain a valid file system (either because the medium was never initialized to contain one or because the file system has become malformed), the device specification has the form

{Node}[DevName]

In this case, the device password must be specified. A *device password* protects a device. It can have a maximum of 12 characters, consisting of all alphanumeric characters plus the period (.) and the hyphen (-).

A *volume name* or a *device name* is a string of characters. A volume name or device name can have a maximum of 12 characters, consisting of all alphanumeric characters, plus the period (.) and the hyphen (-).

Memory Disk

A memory disk is a software memory disk device that behaves like a physical disk.

It is formatted with the same volume control structures (Volume Home Block, Master File Directory, and so forth) that are used to format a physical disk. Access to the memory disk files by file management is accomplished through these same structures.

Memory disks are created using the **Format Disk** command. (For details on memory disk initialization, see the *CTOS System Administration Guide*.)

Unlike a physical disk, a memory disk is volatile memory. For this reason, you should only use one for the storage of temporary files.

Cached Memory Disk

A cached memory disk is a memory disk, the space for which is taken from the file system cache pool. Cache blocks are not allocated for the disk until files are actually written onto it.

Typically you use a cached memory disks as the scratch volume *[Scr]*. You can create a cached memory disk by an entry in the system configuration file, *Config.sys*. (For details, see “Using a Cache Memory Disk” in the *CTOS System Administration Guide*.)

Disk Partitions

On SuperGen Series 5000 and EISA/ISA-bus workstations, the CTOS volume structures are contained in a larger-grained structure called a *disk partition*. A disk with partitions can accommodate more than one operating system environment in separate partitions.

A disk can have a maximum of four partitions. Two are required for CTOS, if a crash dump file is included; otherwise, the CTOS volume structures can be contained in a single partition.

Note: *All of the disk partitions can be used by operating system environments other than CTOS.*

CTOS has no knowledge of the format of the partitions it is not using. Applications running on CTOS, however, can read and modify the CTOS partition(s).

Say, for example, your application needs to access the file system volume home block (VHB). Normally it can obtain the address of the working copy of the VHB by calling the GetVhb operation. GetVhb returns the dynamically up-to-date VHB version. (To ensure system integrity, it is recommended that the duplicate copies of the volume control structures be maintained on a volume.)

If your application detects that the VHB returned is corrupt, it can still retrieve VHB information from the backup copy (used to boot the system).

On a disk without partitions, the boot copy of the VHB starts at logical file address (lfa) 0 from the start of the disk. On a disk with partitions, however, the VHB starts at the lfa of the CTOS partition. To obtain the CTOS partition lfa, your application can call the GetCtosDiskPartition operation. The operation returns the lfa and the partition size. Using the lfa, your application can obtain CTOS information in the usual manner, as volume control structures are in the same relative positions whether or not the disk has partitions.

(For details on how to format a disk with partitions, see the description of the **Format Disk** command in the *CTOS Executive Reference Manual*.)

Disk Management Operations

The disk management operations are described below. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

DismountVolume

Dismounts the specified volume.

Format

Initializes the surface of a floppy disk or other disk media to accommodate fixed-size data sectors. Format is used by the **Format Disk** command.

GetCtosDiskPartition

Returns the lfa from the start of the disk and the size of the CTOS disk partition.

MountVolume

Mounts the volume on the specified disk drive.

QueryDiskGeometry

Returns disk geometry information for the specified device to the memory area provided.

QueryDeviceName

Returns the name entry in the workstation or shared resource processor CPU device name table for each device specified.

QueryDeviceNames

Returns the workstation or shared resource processor CPU device name table.

SetDevParams

Allows the characteristics of the floppy disk controller to be modified.

SetDiskGeometry

Sets the disk geometry for the specified device.

Section 14

Printing Management

Printing management provides a Generic Print System (GPS) to route output to the printer. If GPS is installed, it takes precedence over pre-GPS printing. (For details on pre-GPS printing, see “Programmer’s Notes on the Spooler” in the *CTOS Programming Guide*.)

Generic Print System Components

GPS consists of the following components:

- Queue Manager
- Print Service
- Font Service
- Device Driver

The above components work together to control printing and to handle communication between the application program, the operating system, and the installed printing devices.

GPS is a separate program from the operating system and, as such, is covered comprehensively in separate manuals. (For installation details, see the *Generic Print System Administration Guide*; for programming information, see the *Generic Print System Programming Guide*.)

Interface Considerations

You can choose to request output to a GPS printing device through any of the following interfaces:

- **Sequential access method (SAM).** In accessing SAM directly, you sidestep GPAM's controls but retain device-independence. From a programmer's viewpoint, SAM is the simplest way to print a document. You specify the GPS printer name, and the GPS system services handle all aspects of printing for you. (See "Sequential Access Method.")
- **Generic Print Access Method (GPAM).** GPAM is a device-independent means of adding complex text formatting, such as boldface, text, or graphics, to your output with a minimum of programming effort. GPAM sends its control information to the printing device through SAM. (See "Generic Print Access Method.")
- **Direct GPS request.** At the direct interface level, your program becomes GPS-dependent. This is not the recommended method.

File access methods. See the section entitled "Structured File Access Methods." It is a guide to three high-level I/O interfaces to structured data files.

Section 15

Communications Programming

What is Communications Programming?

This section describes *communications programming* at the device-dependent and the device-independent interface levels. For details and examples of how to use the communications programming operations, see the *CTOS Programming Guide*.

- At the device-dependent level, communications byte streams (SamC) consists of the device-dependent interfaces of the Sequential Access Method (SAM). These interfaces provide greater control through a variety of operations specific to communications needs. SamC is the standard way to access RS-232-C ports in asynchronous mode.
- At the device-independent level, SAM allows your program to send I/O to a variety of devices. Using communication byte streams at this level is described briefly in this section for comparison purposes. (For details, see the section entitled “Sequential Access Method.”)

What SamC Is Used For

SamC is the RS-232-C device-dependent portion of SAM. It is the standard operating system driver for RS-232-C ports (in asynchronous mode). This includes the use of ports for terminals, modems, and serial printers, as well as direct inter-CPU connection.

Using the standard RS-232-C driver frees the applications programmer from having to write interrupt handlers (described in the section entitled “Interrupt Handlers”), buffer management procedures, serial controller chip initialization sequences, and other low-level software.

SamC is intended to be flexible enough to do anything you might need to do with a serial port, except synchronous RS-232-C communication, which is not supported. You can use SamC indirectly, as part of SAM, which preserves device independence (the ability to perform I/O on SamC or a disk file interchangeably, for example). Alternatively, for special needs, you can call SamC directly using its device-dependent interfaces. (SAM does not provide access to all of these interfaces.)

What Programs Use SamC

Programs that accept or internally generate operating system file specifications beginning with *[COMM]* or *[PTR]* use SamC. SamC is linked with the program's run file.

Clients of SamC include the Executive **Copy** command and the spooler (for serial printers).

What Programs Cannot Use SamC

Programs based on a synchronous RS-232-C communications protocol cannot use SamC. Such programs must interface directly with the operating system at a lower level. (For details, see the section entitled "Serial Port Management.")

At the Device-Independent Interface Level

SAM allows you to access communication ports from your program at the level of *OpenByteStream*, *ReadBsRecord*, *WriteBsRecord*, and the other device-independent byte stream operations described in the section entitled "Sequential Access Method." To use the device-independent SAM operations, you must specify a device in your *OpenByteStream* call. (For a list of the device specifications, see "Device/File Specifications," in the section entitled "File Management.")

SAM can be configured to include or exclude support for particular devices. Each device type has a corresponding byte stream. You can choose your own subset of the byte stream types, depending upon your needs and memory requirements.

To use SamC through SAM, it is necessary to have a configuration file for each communications channel. The configuration file specifies options for devices attached to the channel. As an example, separate transmission/receive baud rates may be required. You can use the default configuration file, or you can use the Create Configuration File utility to create or edit configuration files. (For details, see the Create Configuration File utility in the *CTOS Executive Reference Manual*.)

The configuration file supports parallel printer, serial printer, and communications configurations. SamC handles serial printer (*[PTR]*) and communications (*[COMM]*) configurations. You can open both kinds of configuration files with *[COMM]* or *[PTR]* device specifications.

(See the *CTOS Programming Guide* for details.)

At the Device-Dependent Interface Level

The device-dependent interfaces of SamC itself (as distinct from SAM of which it is a part) provide a more powerful and flexible set of services than those available at the level of SAM.

Programs that are distinctly communications oriented (as opposed to programs such as the Executive, which merely use SamC through SAM as it would any other type of byte stream) can take advantage of the SamC services.

SamC also supports operations that are not appropriate for other byte stream types. Programs may supplement SAM by occasionally using SamC interfaces.

Although more complex to use than SAM, SamC comprises a complete set of services and can act as a replacement for SAM (provided communications byte streams and no other device types need be supported). Used in this fashion, SamC is a general-purpose device driver for asynchronous RS-232-C communications. It can form the heart of virtually any communications product except those that use synchronous communications protocols. Both half- and full-duplex communications are supported efficiently with a variety of line control and data editing options. Among other conveniences, using SamC frees you from writing interrupt handlers.

Using the SamC Operations

The SamC operations can be used in various ways. The following paragraphs discuss their use in asynchronous operation, opening a byte stream without a configuration file, dynamically changing configuration file parameters, querying and setting status, and setting auto start for spoolers.

Asynchronous Interface

Because SamC is a subroutine package, you cannot issue asynchronous requests to it as you can with disk or keyboard byte streams, for example. (Asynchronous requests allow the caller to continue executing rather than waiting at an exchange.) For this reason, asynchronous variants of the synchronous interfaces are provided, as follows:

Asynchronous	Synchronous
FillBufferAsyncC	FillBufferC
FlushBufferAsyncC	FlushBufferC
CheckpointBsAsyncC	CheckpointBsC,

Some applications require using asynchronous interfaces. MS-DOS, for example, must be able to initiate FillBufferC (communications input) and FlushBufferC (communications output) operations without the possibility of waiting as a side-effect.

The asynchronous operations include additional parameter options that allow the caller to specify what SamC should do if it needs to wait before the operation can be completed. As an example, one option provides using the Kernel primitive PSend to send a message to a caller-specified exchange when completion becomes possible. (PSend and other Kernel primitives for sending messages are described in detail in the section entitled "Interprocess Communication.")

FillBufferAsyncC provides a way to check the Byte Stream Work Area (BSWA) contents for input without waiting, if no input is there. In the past, SamC users often peeked into the BSWA to see if input characters were waiting. Doing so required knowledge of the BSWA, communications byte stream's private control structure. This is not recommended, however, because the BSWA contents change from release to release.

The AcquireByteStreamC Operation (Low-Level Open)

The OpenByteStream and OpenByteStreamC operations require a configuration file containing the communications line configuration parameters (baud rate and so on). AcquireByteStreamC is a lower-level interface that accepts an in-memory structure corresponding to the configuration file contents. Applications, such as electronic mail, use this interface to open SamC channels, thus avoiding an actual configuration file on disk. (For details, see the *CTOS Programming Guide*.)

AcquireByteStreamC also provides for greater control over the buffer sizes chosen for the receive and transmit queues. Under OpenByteStreamC, the caller supplies a single memory area of a chosen size, which OpenByteStreamC divides up between receive and transmit queues, according to its needs.

Dynamically Changing Parameters

SamC provides a way to query or change configuration parameters without closing and reopening the byte stream. Electronic mail uses this feature to change the baud rate and other parameters without closing the byte stream (and thereby disconnecting an attached modem).

Querying and Setting Status Lines

The RS-232-C standard defines additional status lines that are not used by SamC but may be significant when dealing with modems or special hardware. Communications byte streams provide an interface to access or, where appropriate, to change the state of these lines.

The CheckForOperatorRestartC Operation

The spooler periodically can call the CheckForOperatorRestartC operation to support auto restart on printers. This feature makes it possible for the spooler to restart output in response to

- An operator pressing the On/Offline switch on the printer
- An operator opening and then closing the printer cover (on a printer with no Break switch)

SamC Operations

The SamC operations are described below. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

OpenByteStreamC*

Opens a (*[COMM]* or *[PTR]*) byte stream device pecific to an RS-232-C serial port.

AcquireByteStreamC

Is a substitute for *OpenByteStreamC* that does not require a configuration file on disk and offers more flexibility.

FillBufferC*

Reads characters from the receive queue that have been received at the serial port.

FillBufferAsyncC

Is the asynchronous form of *FillBufferC*.

FlushBufferC*

Writes characters to the transmit queue, where they will be output to the serial port.

FlushBufferAsyncC

Is the asynchronous form of *FlushBufferC*.

DiscardInputBsC

Discards any characters in the receive queue.

DiscardOutputBsC

Discards any characters in the transmit queue.

*This operation is the communications byte stream variant of a device-dependent SAM operation. (See the section entitled "Device-Dependent SAM.")

SetImageModeC*

Sets normal, image, or binary mode for [COMM] and [PTR] byte streams device specific to RS-232-C serial ports.

ReadByteStreamParameterC

Reports the current value of the specified communications line parameter.

WriteByteStreamParameterC

Modifies the value of the specified communications line parameter.

ReadStatusC

Reads the values of the specified status bits.

WriteStatusC

Writes to the specified communications lines status bits, changing the condition of the corresponding status lines.

CheckForOperatorRestartC

Checks for an operator signal to restart the printer.

SendBreakC

Sends a break signal on the communications line previously opened under the sequential access method.

CheckpointBsC*

Waits until all characters previously written to the byte stream have been physically output from the serial port.

*This operation is the communications byte stream variant of a device-dependent SAM operation. (See the section entitled "Device-Dependent SAM.")

CheckpointBsAsyncC

Asynchronous form of CheckPointBsC that can be used to perform the CheckPointBsC function when the caller does not want its process to wait.

ReleaseByteStreamC*

Stops all receive and transmit operations on a serial byte stream, making the serial port available for use by other users again.

*This operation is the communications byte stream variant of a device-dependent SAM operation. (See the section entitled "Device-Dependent SAM.")

Section 16

Serial Port Management

Access Below the Byte Stream Level (CommLine)

Serial port management pertains to communications programming at the *serial port* interface level. This is a level below the SamC interfaces. (See the section entitled “Communications Programming” for SamC details.)

SamC does not support a serial communications controller in synchronous mode. To write a program that uses a synchronous communication protocol, it is necessary to interface directly with the operating system at a level below SamC.

The following operations are part of the operating system’s support for serial ports:

- InitCommLine
- ResetCommLine
- ChangeCommLineBaudRate
- TerminateCommLine
- ReadCommLineStatus
- WriteCommLineStatus

SamC calls the serial port operations on behalf of SamC clients. (For details on how to use the operations, see *CTOS/Open Programming Practices and Standards*.) Note that the operations are compatible with raw interrupt handlers in protected mode. (See the section entitled “Interrupt Handlers,” for details on raw interrupt handlers.)

Serial Port Operations

All Unisys synchronous communications products that do not use the SamC level of interface use the serial port operations. Although it is the intent of these operations to reduce hardware dependencies, it is important that the programmer be aware that hardware specific details cannot be entirely eliminated below the level of byte streams. As an example, the InitCommLine operation can return information about the current hardware, such as whether it employs an RS-232-C, X.25, or V.35 communications interface. However, the programmer is responsible for knowing the impact of a particular serial controller/interface combination. (For details, see the hardware documentation.)

Using InitCommLine

The InitCommLine operation assigns the caller to a physical channel on a serial communications controller. InitCommLine does this by parsing the device specification passed to it. An application should treat the specification as an uninterpreted string so that it can continue to work when new hardware modules (with new forms of file specifications) are introduced.

InitCommLine returns information to the caller in the communications line return block (CLRB). Included in the information returned are the addresses of a control register and a data register for the channel. The contents of the CLRB can vary, depending on the hardware. (For details, see the description of the InitCommLine operation in the *CTOS Procedural Interface Reference Manual*.)

If the communications controller is shared (has two channels, A and B), certain operations are always performed on one of the channels that affect both. InitCommLine performs these functions for the caller. After an interrupt, InitCommLine resets the controller.

The caller still must perform some operations directly on the channel using the register addresses. `InitCommLine` does not even fully initialize the channel (although it does reset it), since it is not provided all of the initialization parameters. Note that the only parameters supplied to `InitCommLine` are those dealing with external hardware (outside the serial controller). This hardware (baud rate timers and external control registers) is `InitCommLine`'s responsibility because it varies from machine to machine. The controller, however, is invariant: all hardware uses the same (or software-equivalent) serial controller-type chips.

Using `ResetCommLine`

An application cannot issue `ResetCommLine`, or any other operation, until it has successfully completed an `InitCommLine` operation for that channel. The argument to `ResetCommLine` is a handle returned by `InitCommLine`.

`InitCommLine` acquires the channel for the caller (and resets it so the caller has a chance to initialize it to its specifications before starting to take interrupts). `ResetCommLine` gives the channel back to the operating system, making it available for other users and freeing the caller from the responsibility of servicing interrupts from it. Thus, the `InitCommLine` and `ResetCommLine` operations are logical parentheses, like `OpenFile` and `CloseFile`, for a serial port.

Using `ChangeCommLineBaudRate`

`ChangeCommLineBaudRate` is used to change `InitCommLine` baud rate parameters dynamically.

`SamC` uses this interface to implement its `WriteByteStreamParameterC` call. `SamC` clients should use `WriteByteStreamParameterC` to modify the baud rate(s) dynamically. They should not use the `ChangeCommLineBaudRate` operation directly.

Using ReadCommLineStatus

ReadCommLineStatus allows certain serial communications signals, whose functions are not defined by the serial controller, to be queried by the application program in machine-independent fashion.

SamC uses this interface to implement its **ReadStatusC** call.

ReadStatusC is the way communications byte stream clients should query the status lines. They should not use **ReadCommLineStatus** directly.

Using WriteCommLineStatus

WriteCommLineStatus allows certain serial communications signals, whose functions are not defined by the serial controller, to be raised or lowered by the application program in machine-independent fashion.

SamC uses this interface to implement its **WriteStatusC** call.

WriteStatusC is the way communications byte stream clients should set or clear the status lines. They should not use **WriteCommLineStatus** directly.

Serial Port Management Operations

The serial port operations are described below. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

InitCommLine

Allocates a serial port to the user and specifies how interrupts from the port will be serviced.

ReadCommLineStatus

Reads values to the specified status bits.

WriteCommLineStatus

Writes to the specified status bits, changing the condition of the corresponding status lines.

ChangeCommLineBaudRate

Reinitializes the specified baud rate timer(s).

ResetCommLine

Makes the specified serial port available for use again.

LockIn

Allows a program to read from the serial I/O port. LockIn is essential on certain types of workstation hardware because of the timing functions it performs.

LockOut

Allows a program to write to the serial I/O port. LockOut is essential on certain types of workstation hardware because of the timing functions it performs.

GetCommLineDmaStatus

Returns the number of bytes transferred during the current CommLineDma operations (TransmitCommLineDma or ReceiveCommLineDma) for both the transmit and receive DMA channels.

ReceiveCommLineDma

Sets up the DMA controller to transfer a specified number of bytes from the communications channel to the specified memory buffer.

TransmitCommLineDma

Sets up the DMA controller to transfer a specified number of bytes from the specified memory buffer to the communications channel.

Section 17

SRP Terminal Management

Program Access to Ports

The *SRP terminal management* operations are programming interfaces to shared resource processor (SRP) terminals attached to all ports. However, there are certain ports that can be accessed only by these interfaces. Figure 17-1 shows the relationships of ports to access methods for shared resource processors and workstations.

Note: *Although they are still supported, the shared resource processor TP and CP boards are not commonly used with CTOS/XE protected mode boards.*

The communications programming operations described in the section entitled “Communications Programming” are at the same interface level as the SRP terminal management operations.

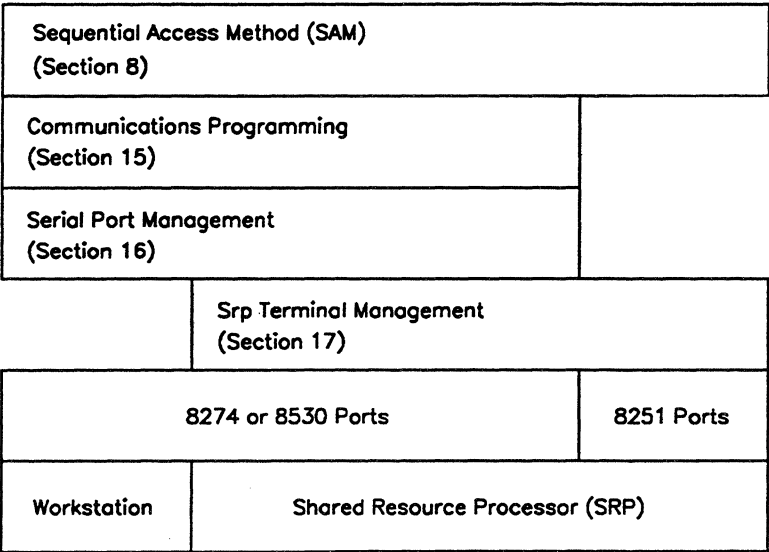
Figure 17-1 indicates that you can access the 8274 or 8530 ports using either of these operation groups. You would most likely use the SRP terminal operations if you need to access the 8251 ports.

At a level farther away from the hardware, you can use the device-independent Sequential Access Method (SAM) operations to access all ports.

For details on the other interfaces illustrated, see the following sections:

- “Sequential Access Method”
- “Communications Programming”
- “Serial Port Management”

Figure 17-1. Ports/Access Methods Relationship



557.17-1

SRP Terminal Management Operations

The SRP terminal management operations are described below. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

OpenTerminal

Initiates the use of a specified port on either a Cluster Processor (CP) or a Terminal Processor (TP).

ReadTerminal

Reads data from one of the asynchronous ports on a CP or a TP.

CloseTerminal

Indicates that the requesting process (client) is finished with a port.

SetTerminal

Performs out-of-band functions on a port.

WhereTerminalBuffer

Locates the terminal output buffer.

DrainTerminalOutput

Ensures an empty output buffer.

Section 18

SCSI Device Management

What is SCSI Management?

SCSI (pronounced “scuzzie”) is the Small Computer Systems Interface, an American National Standard for the interconnection of computers with peripheral devices such as disk drives, tape drives, and printers. The SCSI standard is defined in American National Standards Institute (ANSI) document X3.131-1986. Extensions to the standard are known as SCSI-2.

The SCSI Manager is a system service supported on protected mode versions of the operating system. (See Appendix A, “Operating System Features” a list of the supporting systems.) It provides access to any SCSI device connected to a workstation or shared resource processor that has SCSI capability. SCSI devices may be the familiar peripherals previously mentioned, they may be new types of devices currently under development [for example, document scanners, media-changers (jukeboxes), and optical media such as CD-ROM or WORM], or they may be as yet unanticipated devices. The SCSI Manager is written to comply with both the SCSI standard and the SCSI-2 extensions.

The SCSI standard consists of layers. Each layer defines certain requirements for implementation, as described below:

- The *physical* (and *electrical*) layer defines (among other things) the form and shape of the SCSI connectors, the voltage levels, and the timing relationships of the electrical signals used in SCSI.
- The *transport protocol* layer defines how logical connections are established between SCSI devices; how commands, data, and status are transferred between devices; and how the integrity of this information is assured. The transport protocol layer, however, does not address the meaning of the commands or data it manages. This is the responsibility of the logical session layer.

- The *logical session* layer defines the meanings of commands sent to SCSI devices and the actions at the devices as a result of the commands and the data (transferred to or from the devices) accompanying the commands.

Of these layers, the physical layer is embodied in the hardware, and the logical session layer is the domain of your application program or system service written to control a particular SCSI device. Between the foregoing layers is the SCSI Manager. It controls the transport protocol layer by acting upon the commands received from your software and by programming the hardware to transport these commands to the SCSI device.

The *SCSI bus* is the physical cable that connects the computer and all the SCSI devices. It can be viewed as the communications highway between your software and the SCSI devices.

As transport protocol controller, the SCSI Manager is responsible for traffic over the SCSI bus. The SCSI Manager establishes the virtual circuit connection (or path) between a user program and a SCSI device and oversees the reliable transportation of commands and data between the program and the device. The SCSI Manager knows nothing of the content of the information sent back and forth. It does not know the nature of the commands a program issues to the device or the resulting actions of these commands at the device. It simply oversees the harmonious use of a SCSI bus by all programs accessing devices connected to it. In this way, each user program can concentrate on its own independent communication with a SCSI device without regard to how or when a bus is being used by other programs.

The remainder of this section describes the capabilities of the SCSI Manager in greater detail and provides examples of typical SCSI Manager uses. If you are the developer of an application to control a SCSI device, you should be familiar with the following documentation:

- The SCSI Manager concepts presented in this section
- The SCSI operations described in the *CTOS Procedural Interface Reference Manual*
- The documentation provided by the manufacturer of the SCSI device you want to control
- Portions of the SCSI standard

In addition, you may need to refer to your release documentation and to the hardware manuals for the hardware products you are using. Because of hardware limitations, some systems cannot support all the SCSI Manager features.

SCSI Management Terminology

Brief definitions of terms used in this section are presented in Table 18-1. You may wish to refer back to this section as you read. Some of the terms are more meaningful in the context with which they are used.

Table 18-1. SCSI Device Management Terms

Term	Definition
Command descriptor block (CDB)	The data structure used to communicate requests from an application program to a SCSI device.
Hostadapter	A hardware device interposed between a computer system and a SCSI bus. The device usually performs the lower layers of the SCSI protocol and normally operates in the initiator role.
Initiator	A SCSI device (usually a host computer system) that requests an I/O process to be performed by another SCSI device (a target).
Logical unit	A physical or virtual peripheral device addressable through a SCSI target.
Logical unit number (LUN)	An encoded 3 bit identifier of a logical unit.
Peripheral device	A physical device that can be attached to a SCSI device which connects to the SCSI bus. The peripheral device and the SCSI device (peripheral controller) may be physically packaged together (embedded). Often a SCSI device maps to one logical unit, but this is not always the case.

continued

Table 18-1. SCSI Device Management Terms (cont.)

Term	Definition
SCSI (target) ID	The encoded representation of the unique address (0 through 7) assigned to a SCSI device. The address is normally assigned and set in the SCSI device during system installation.
SCSI device	A host computer adapter, peripheral controller, or an intelligent peripheral that can be attached to the SCSI bus.
Target status	One byte of information sent from a SCSI target to an initiator, upon completion of each command.

Overview of the SCSI Manager's Capabilities

The SCSI Manager provides many facilities to the user. The important capabilities are summarized below.

Convenience

- **Hardware independence:** Regardless of the SCSI host adapter hardware, the same logical interface is provided to communicate with all SCSI devices.
- **Resource management:** The SCSI Manager permits many users to share access to the SCSI bus and/or SCSI devices without conflict. The usage load on the SCSI bus is balanced to provide equitable access among all the users.
- **Protocol management:** An application that uses the SCSI Manager need only concern itself with the logical characteristics of the device it intends to control; the intricacies of the SCSI protocol in transferring commands and data are transparent to the application.

Efficiency

- **Data throughput:** Within the hardware limitations of the SCSI host adapter and the SCSI device(s), the logical connection between the application program and the SCSI device is optimized to use DMA, synchronous data transfers, and linked commands. This maximizes sustained data transfer rates over the SCSI bus.
- **Bus utilization:** Many commands may be in progress at many different SCSI devices; the SCSI bus (common medium for the transfer of all commands and data between the SCSI host adapter and the peripherals) is efficiently shared by multiplexing access among all devices.

Reliability

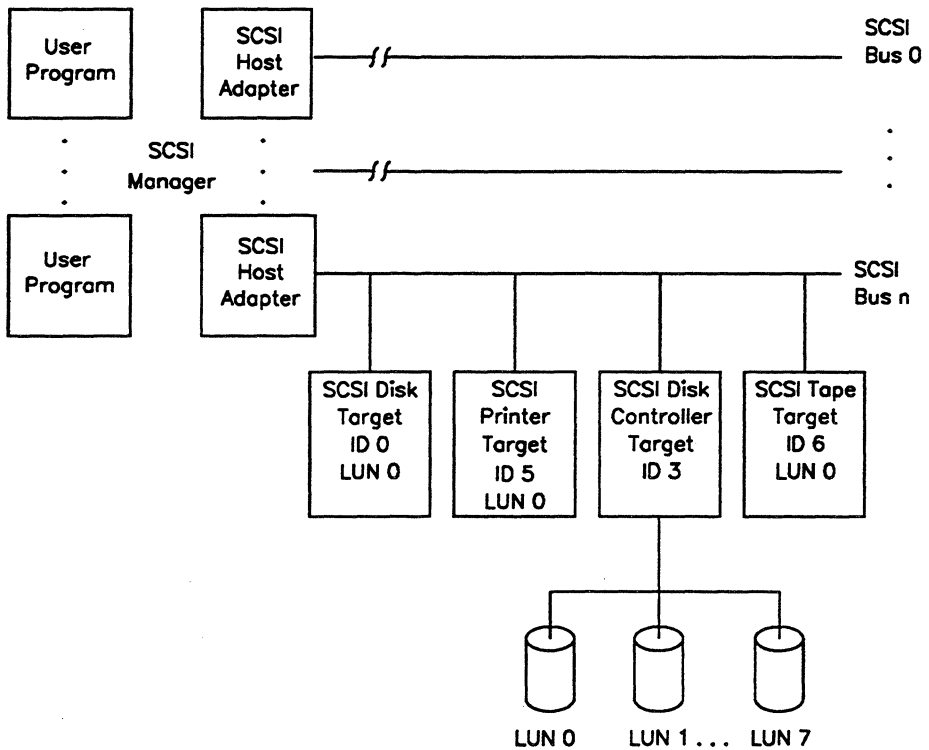
- **Data integrity:** SCSI protocol detects errors in the normal transmission of commands and data over the SCSI bus. Some of the errors can be corrected by the SCSI Manager itself, while others are reported to the user so that appropriate recovery may be initiated.
- **Password protection:** The SCSI Manager allows three categories of access (exclusive, shared, and peek mode) to SCSI devices, and provides for password verification to enforce access rights. (These access modes are distinguished from similar modes used by the file management system.)

Creating a SCSI Path

Before your program can attempt operations with a SCSI device, it must call the `ScsiOpenPath` operation to establish a virtual circuit connection. The connection (or *path*) defines the relationship between a user program and the SCSI device and is maintained in the control structures of the SCSI Manager.

Figure 18-1 illustrates the relationships between user programs, the SCSI Manager, the SCSI host adapter, and the various SCSI devices connected to the SCSI bus. In the figure, the starting point of a path is a SCSI host adapter. Its ending point is a SCSI device. Each SCSI device is identified by a target ID and a logical unit number (LUN), as shown.

Figure 18-1. Sample SCSI Configuration



557.18-1

Although the figure depicts a single physical connection (SCSI bus) between a given SCSI host adapter and its associated SCSI devices, the SCSI Manager can establish and manage many different paths between user programs and SCSI devices. Each unique path is created by a user program calling the `ScsiOpenPath` operation, the parameters to which are described next in this section. Because of the special significance of the password and access mode to `ScsiOpenPath`, these parameters are discussed at length in “SCSI Access Modes” and “SCSI Passwords,” as well as in “Example of Using the SCSI Manager.”

Configuring a SCSI Manager

Before using `ScsiOpenPath`, if you want to use a SCSI Manager name other than the default `[SCSI]`, you must add the name to the system configuration file at the originating processor. Any name may be specified. Typical examples are `[SCSI]`, `[SCSI0]`, and `[SCSI1]`. If the SCSI Manager name is not provided in the call to `ScsiOpenPath`, no routing is performed. To query the name of the SCSI Manager at the specified processor, your program can call the `ScsiManagerNameQuery` operation.

Defining a Unique Path

In the call to `ScsiOpenPath`, your program must identify the unique starting point (SCSI Manager and host adapter) and ending point (SCSI device) of the path. The SCSI Manager is identified by the name provided in the call to `ScsiOpenPath`, as previously described. The host adapter and device are identified by numbers, as described below.

On systems with only one SCSI bus, there is one host adapter and the adapter number is 0. If, however, a system has more than one SCSI bus, there is a SCSI host adapter corresponding to each bus. For example, a Series 386i or B39 workstation with a SCSI Upgrade Module has two SCSI host adapters: one is internal to the workstation and the other, to the Upgrade Module. To reference a SCSI device connected to the Upgrade Module's SCSI bus, your program must specify host adapter number 1 (rather than 0) in the call to `ScsiOpenPath`. Similarly, the GP+SI processor board in a shared resource processor system has two SCSI host adapters, 0 and 1. (For further information, see your hardware documentation.)

At the end of the SCSI path, the SCSI device is uniquely identified by two numbers, a target ID and LUN. Each of these numbers is in the range of 0 through 7. Although each target ID can be associated with up to 8 LUNs, in common practice most of the SCSI peripherals currently in use support only one LUN (LUN number 0) per target ID.

Specifying Path Parameters

In addition to defining the unique path, optional parameters to the `ScsiOpenPath` operation (commonly called path parameters) allow your program to control certain operating characteristics of a path. Path parameters are unique for each path opened to a SCSI device.

The values of the path parameters are specified in the path parameters block structure. (For the format of the path parameter block, see the description of `PIRScsiOpenPath` in the *CTOS Procedural Interface Reference Manual*.) Through values specified, the following path characteristics can be defined:

- **Disconnection permission:** By default, SCSI devices are permitted to disconnect from the SCSI bus while they are processing a command. This frees the SCSI bus for use by other devices while processing is taking place. Allowing disconnection is usually the most efficient way to use the SCSI bus. If, however, it is important to your application that the SCSI device remain connected to the bus for the duration of command execution, you can specify that disconnection not occur.
- **Default timeout:** All SCSI operations are required to complete within a specified time limit, or error recovery procedures are initiated. You may specify the time limit explicitly for each SCSI operation. Otherwise, the default timeout set in the path parameters block is used.

If your program provides a 0 value for any field in the path parameters block, the default value (most common parameter interpretation) is used. Omitting the path parameter block entirely (that is, specifying NULL for the address of the path parameter block or a block length of 0) causes the default value to be used for every parameter field. (This is shown in “Example of Using the SCSI Manager.”)

Changing and Querying Path Parameters

Although the `ScsiOpenPath` operation allows you to specify path parameters at the time a SCSI path is established, your program may query and change the parameters later by calling the `ScsiQueryPathParameters` and `ScsiSetPathParameters` operations, respectively.

Path parameters that may not be set by the caller may be obtained by calling the `ScsiQueryPathParameters` operation. One such read-only parameter indicates whether synchronous data transfer is in use with the SCSI device. (For a list of all the read-only parameters, see the description of the `ScsiOpenPath` operation in the *CTOS Procedural Interface Reference Manual*.)

Using the Path Handle

In addition to the parameters described thus far, your program can provide an access mode and a password to `ScsiOpenPath`. (See “SCSI Access Modes” and “SCSI Passwords.”) For the moment, however, assume that you have successfully opened a path to the SCSI device you intend to control. Upon a successful open, `ScsiOpenPath` returns a path handle to your program.

A *path handle* is a 16 bit, unsigned number that is valid for future references to the SCSI device you have uniquely identified. All other SCSI operations require that you use the path handle to indicate which SCSI device you intend to address. When your program has completed operations with the SCSI device, it must call the `ScsiClosePath` operation to return the path handle and free any SCSI Manager resources that were allocated to your path.

SCSI Access Modes

The SCSI Manager recognizes three different modes of access to SCSI devices: exclusive, shared, and peek. These modes are established by the `ScsiOpenPath` operation and represent different privilege levels of access. More precisely, these modes are

- **Exclusive mode (mx).** This mode is permitted to only one user of a SCSI device at a time. Your program may obtain exclusive access only if one of the following is TRUE:

1. No other paths are open to the SCSI device.
2. The only paths that are open were opened in peek mode.

After you have established exclusive access, any subsequent `ScsiOpenPath` attempts for the device are denied until you have closed the path with exclusive access. When exclusive access to a device is granted, the SCSI Manager invalidates all path handles that were established in peek mode (see below).

- **Shared mode (ms).** This mode is permitted to any number of users of a SCSI device. Users of paths established in peek mode are undisturbed by paths subsequently established in shared mode and new paths in either shared or peek mode may be established after the first shared mode path is created.
- **Peek mode (mp).** This is a special mode available for users that want to access a SCSI device but, at the same time, do not want to stand in the way of any other user's attempt to create a path to the device in exclusive mode. Paths created in peek mode may be invalidated by the SCSI Manager any time another user opens an exclusive path to the SCSI device. The first notification the user of a peek mode path has of this event is the return of a status code indicating an invalid path handle when any SCSI operation (including `ScsiClosePath`) is attempted with the handle.

Although there are similarities, the SCSI Manager access modes are not to be confused with the file management access modes (modify, peek, and read). Unlike the file management modes, all SCSI Manager access modes grant *unrestricted permission* to modify data stored by the SCSI device. Because of this, to have sole responsibility for the integrity of information on a SCSI device, it is recommended that the user program do either of the following:

- Establish a path to the device in exclusive mode. In this mode, no other user can send potentially disruptive SCSI commands to the device.
- Be the first to establish a path to the device in shared mode with a password. By doing so, the only other programs that may access the device are those with knowledge of the password.

SCSI Passwords

The necessary companion to the access mode is the password associated with a SCSI device. A password is used to validate the creation of a SCSI path in one of the access modes. With the exception of direct-access devices (discussed next in this section), most SCSI devices have no nonvolatile medium on which password information may be stored and retained during periods when the computer system or the SCSI device is powered off. Because of this, passwords within the SCSI Manager are not permanent (they exist only in memory and are not retained when the system is powered off) and follow rules of behavior that differ radically from file management passwords.

A password for a SCSI device is initialized when your program establishes a path to a SCSI device for which no other paths (except peek mode paths) exist. In this case, the password supplied in the `ScsiOpenPath` operation becomes the password for the SCSI device. The password remains in effect until your program closes the path that initialized the SCSI device password. Any subsequent attempts to establish a path to the SCSI device by your program (or any other user program) require the same password be provided regardless of the access mode requested. Otherwise the attempt is denied.

File System and the SCSI Manager

If you intend to use the SCSI Manager to control or give commands to a SCSI device that is also a file system device, you need to understand the relationship between the file system and the SCSI Manager. With its need for password protection and enforcement of access privileges to system volumes on direct-access devices, the file system is a special SCSI Manager user. As such, it has privileged, internal methods of communication with the SCSI Manager.

When an operating system (a workstation or SRP processor board) is booted, it performs the necessary hardware and software initialization. For the file system, this includes determining whether peripheral devices *might* contain valid system volumes followed by attempting to mount system volumes on all such devices. Direct-access devices that predate SCSI (for example, ST-506 Winchester disks on workstations and shared resource processor file processors or SMD disks on shared resource processor data processors) were less sophisticated and less subject to varying configurations. As an example, the disk controller hardware on a file processor can support up to four disk drives. The file system simply assumes that four disks are present and creates the necessary control structures (device control blockss) for them. SCSI devices, however, require a different approach as outlined below.

During the file system hardware initialization phase, the SCSI bus is scanned to locate any devices identifying themselves as direct-access devices. (See "Example of Using the SCSI Manager.") These devices automatically fall under the control of the file system and are allocated device control blocks (DCBs). When a DCB is created, the file system calls `ScsiOpenPath` to create a path handle for subsequent references to the path. To allow utility programs access to the device through the SCSI Manager, the path is opened in shared mode. The *device password* from the DCB is used to establish the initial password for the SCSI device.

Once file system hardware initialization is complete, an attempt is made to mount valid system volumes on all direct-access devices with DCBs. If a system volume is present at the device, the password for the SCSI device is changed from the device password to the volume password. The access mode remains shared mode; any other user that wants to perform SCSI operations on the device (even nondestructive operations, such as requesting the medium defect information from the SCSI disk) must supply the volume password to open a SCSI path to the device.

Some SCSI disks that have system volumes may not be powered on when a system is booted. To allocate the necessary control structures, these devices must be described in the system configuration file. The configuration file entry describes the physical location (which host adapter) and the logical location (target ID and LUN) of the SCSI disk and associates a device name and device password, as shown in the example entry below:

```
:MassStorage:(Class=SCSI, Adapter=0, Target=3, LUN=0, Device=D0,  
Password=D0)
```

An entry like this guarantees that a DCB is allocated for a SCSI disk when the system is booted. Whether or not the SCSI disk is powered on, the DCB ensures that the system volume is properly mounted. As previously described, the password established for the SCSI disk is the *device password* from the configuration file entry.

When a SCSI disk is powered on after system boot, the file system's Automatic Volume Recognition (AVR) feature recognizes that the disk is ready and attempts to mount a system volume. If the disk contains a valid system volume, AVR mounts it and the password for the SCSI device is changed to the *volume password*, as described earlier.

When a SCSI disk is powered off, the AVR also recognizes this condition and dismounts the system volume accordingly. At this point, the password for the SCSI device reverts back to the *device password*.

SCSI Command Structure

Once you have used the `ScsiOpenPath` operation to establish a path to a SCSI device, you need to instruct the device to carry out the intended operations. As previously mentioned, SCSI is essentially a communications protocol. It allows your program to perform the following function sequence:

1. Send commands to devices.
2. Send or receive data.
3. Receive status at the completion of commands.

These different functions are segregated on the SCSI bus by means of the *information transfer phases* described in Table 18-2.

Table 18-2. Information Transfer Phases

Transfer Phase	Description
DATA OUT	Permits the transfer of data (for example, a record to be written on a tape or disk) from the host adapter to the SCSI device.
DATA IN	Permits the transfer of data (for example, a record read from a tape or disk) from the SCSI device to the host adapter.
COMMAND	Transfers a 6, 10, or 12 byte command descriptor block (CDB) from the host adapter to the SCSI device. The information in the CDB describes the command to be performed by the SCSI device (either a command common to all SCSI devices or a device-specific command) and often specifies the transfer length of the DATA IN or DATA OUT phase that is expected to follow the command.
STATUS	Allows the SCSI device to send one byte of status information at the completion of a command to the host adapter.
MESSAGE IN	Transfers one or more message bytes from the SCSI device to the host adapter.
MESSAGE OUT	Transfers one or more message bytes from the host adapter to the SCSI device.

How and when the SCSI device is permitted to change the information-transfer phases is beyond the scope of this section. (For details, see the SCSI standards.) The typical order of information for a SCSI command is as follows:

1. Once a connection is established to the SCSI device, the host adapter and the SCSI device are the exclusive users of the SCSI bus. The SCSI device requests a Command phase and receives all 6, 10, or 12 bytes of command information.
2. Depending on the direction of data transfer specified by the command, the SCSI device then requests either a Data Out (for example, a Write command) or a Data In (for example, a Read command) phase and transfers information up to the maximum transfer length specified in the CDB.
3. A data phase is optional. Some commands, such as Test Unit Ready, do not require a data phase and only return status information.
4. After all the data has been transferred, the SCSI device requests a STATUS phase and sends one byte of status information.
5. Finally, the SCSI device requests a Message In phase to send a message that indicates that the command is finished. At this point, the SCSI device releases the SCSI bus (disconnects), and the SCSI bus becomes available for other devices to use.

In the preceding sequence of information-transfer phases, the SCSI device is free to disconnect from the SCSI bus at any point and to later reconnect to continue the processing of the command. (Note that disconnection of a path may not be allowed, if so specified in the path parameters. For details, see “Specifying Path Parameters.”) If, for example, a Read command is sent to a SCSI disk and the data requested is not already in an internal buffer at the SCSI disk, an appreciable amount of time may pass (because of rotational latency) before the data has been read from the medium and is ready to be transferred over the SCSI bus. In such a case, many SCSI devices disconnect immediately after the command phase and later reconnect for the Data In phase. Until the data is ready, the SCSI bus is available for use by other SCSI devices.

When the SCSI operation completes, a status code, the count of bytes actually transferred in a data phase, and the target status are returned to your program. Your program needs to examine both the status code and the target status to determine the success or failure of the SCSI command. A nonzero value for either indicates an error condition. (For details on error conditions that can occur, see “Error Conditions and SCSI Sense Data.”)

In summary, to use the SCSI command structure effectively in controlling a SCSI device, your program must build the appropriate command descriptor blocks in memory and use the `ScsiCdbDataIn` and the `ScsiCdbDataOut` operations to transfer these commands to the device. (To determine how specific commands are to be executed when received by the device, consult the manufacturer’s reference manual for the SCSI device.) When data is associated with a command, the `ScsiCdbDataIn` and `ScsiCdbDataOut` operations provide an optional in-memory buffer from which the data is taken or to which the data is transferred.

Note: *To determine the success or failure of a command sent to a SCSI device, check both the completion status of the command and the actual count of data transferred.*

Error Conditions and SCSI Sense Data

Because the SCSI Manager does not know the characteristics of all the devices with which it may communicate, it is necessary for your program to check two levels of error status to determine the success or failure of a SCSI operation.

First, check the status code returned from the SCSI operation. A 0 value indicates that communications were successfully completed with the device. Then, check the status to see if it, too, is valid (is 0). Only if both codes are 0 can you assume that the operation has completed without error.

The nonzero status code returned by a SCSI operation usually indicates a protocol failure during communication with the SCSI device. In this case, your program (perhaps in conjunction with the operator) might initiate appropriate reset and recovery actions for the device. In some situations, certain status codes are expected. If, for example, your program attempts a SCSI operation with a device that you have not yet ascertained is present on the SCSI bus, status code 384 ("SCSI select timeout") is normal if there is no such device or the device is not powered on. Status codes returned by `ScsiOpenPath`, such as status code 219 ("Access denied") or status code 393 ("Invalid LUN"), indicate logical problems in establishing the path to the device rather than SCSI protocol failures. (For details on SCSI status codes, see the *CTOS Status Codes Reference Manual*.)

If the status code returned is 0, also examine the target status returned at `pStatusRet` in the `ScsiCdbDataIn` or `ScsiCdbDataOut` operation. SCSI devices normally return 0 (Good) in this status byte if the command executed successfully. If, however, an exception condition arises, a target status of 2 (Check Condition) is returned instead. Check Condition means that the caller needs to determine the cause of what went wrong in more detail. Upon the return of this status, the `ScsiCdbDataIn` and `ScsiCdbDataOut` operations automatically save the extended sense information from the target in the buffer provided. (See the description of the `pExtSenseRet` and `sExtSenseMax` parameters in the *CTOS Procedural Interface Reference Manual*.) Into this buffer the SCSI Manager transfers all the information available from the target as to the nature, cause, and location of the unexpected condition.

If Check Condition is returned by one command and the next command issued to the SCSI device is *not* a Request Sense command, the sense data is cleared and execution of the new command begins. (Consult the SCSI standard for an exact definition of the conditions under which relevant sense data is retained.) Saving the sense data in the buffer provided to `ScsiCdbDataIn` and `ScsiCdbDataOut` ensures that the data will not be lost if another program needs to use the SCSI device.

If the SCSI path was opened in exclusive mode, your program may also obtain sense data by calling the `ScsiRequestSense` operation or by constructing a Request Sense command descriptor block and using `ScsiCdbDataIn` to retrieve the data.

The meaning of the first 18 bytes of SCSI sense data is defined by the SCSI-2 standard. All SCSI-2 devices are required to return at least this much information. In practice, many of the SCSI-1 devices also are capable of returning this quantity and format of sense data. (Consult the manufacturer's reference manual for the SCSI device to determine what kind and how much sense data you can expect.)

The sense data is structured hierarchically. A broadly defined sense key specifies the generic error class, while two more fields, the *additional sense code* and the *additional sense code qualifier*, further refine the level of detail. The meanings of these additional fields may vary considerably from one device class to another and even from one vendor's device to another's within the same device class. The meanings of the sense key, however, are commonly defined as shown in Table 18-3.

A system service or application program that displays the sense key names shown above either through operator interaction or informative messages is probably adequate for day-to-day operations. For troubleshooting, however, it is recommended that a program log or display all of the sense data returned by the Request Sense command.

Table 18-3. Sense Keys

Code	Error	Meaning
00h	NO SENSE	There is no specific sense key information to be reported for the designated logical unit. This means a successful command.
01h	RECOVERED ERROR	The last command completed successfully with some recovery action performed by the SCSI device. It may be possible to determine details from the additional sense bytes and the information field.
02h	NOT READY	The addressed logical unit cannot be accessed. Operator intervention may be required to correct this condition.
03h	MEDIUM ERROR	The last command terminated with a nonrecovered error condition likely caused by a flaw in the medium or an error in the recorded data.
04h	HARDWARE ERROR	The target detected a nonrecoverable hardware failure while performing the last command or during a self test.
06h	UNIT ATTENTION	The SCSI device has been reset. The removable medium may have been removed, or other operating characteristics of the device may have changed.
07h	DATA PROTECT	A read or write command to the medium was attempted but not performed on a protected data block.
08h	BLANK CHECK	A write-once device or a sequential access device encountered a blank medium or end-of-data while reading, or a write-once device encountered a nonblank medium while writing.
09h	VENDOR SPECIFIC	
0Ah	COPY ABORTED	A Copy, Compare, or Copy And Verify command was aborted because of an error condition at the source, destination, or both devices.

continued

Table 18-3. Sense Keys (cont.)

Code	Error	Meaning
0Bh	ABORTED COMMAND	The last command was aborted by the target. The initiator may be able to recover by issuing the command again.
0Ch	EQUAL	A Search Data command has completed after satisfying an equal comparison.
0Dh	VOLUME OVERFLOW	A buffered peripheral device has reached the end of the partition, and data may remain in the buffer that has not been written to the medium.
0Eh	MISCOMPARE	The source data provided by the initiator did not match the data read from the medium.
0Fh	RESERVED	

SCSI Manager Target Mode

All of the preceding discussions have concerned the role of the SCSI Manager as a SCSI *initiator*. In this mode, the SCSI Manager functions to convey commands and data to SCSI devices (*targets*) on the SCSI bus. The SCSI Manager is also capable of operating as a SCSI *target* itself. That is, other initiators connected to the SCSI bus can select the SCSI Manager as if it were a peripheral device and can send commands and data to it. This is called *SCSI Manager target mode*.

For additional information on SCSI Manager target mode, see the *CTOS Programming Guide*. You should also consult the documentation on processor devices in the SCSI-2 standard.

When the SCSI Manager functions as a target device, it acts as a processor device according to the definitions for processor devices in the SCSI-2 standard. The SCSI Manager target mode accepts a limited number of SCSI commands, namely

- Inquiry
- Test Unit Ready
- Request Sense
- Receive
- Send
- Send Diagnostic

These commands are sufficient for the exchange of data with other initiators connected to the SCSI bus. A typical use of the SCSI Manager target mode might be to establish a high-speed, 5 Mb/second Very Local Area Network (VLAN) with other computer systems located nearby. When the distances to be traversed are not too far, SCSI can be a less expensive alternative to other LANs such as Ethernet.

Automatic Target Mode Functions

The SCSI Manager performs some target mode functions automatically, even before any target mode operation requests have been issued by a program. When Test Unit Ready, Inquiry, Request Sense, and Send Diagnostic are sent by another initiator, the SCSI Manager always generates an appropriate response to indicate that a SCSI device in the processor class is present but is not yet ready to receive or transmit data.

Explicitly Enabling Target Mode Functions

To explicitly enable target mode functions for one of the eight LUNs (0 through 7) available in target mode, a program must use the ScsiOpenPath operation to create a connection with the desired LUN. The ScsiOpenPath operation works as previously described except that the *targetId* parameter is set to 0FFh. This indicates that target mode operations are to be performed with the returned path handle and that the path handle refers to a LUN of the SCSI Manager processor device, not to a target ID and LUN combination of an externally connected SCSI device. Once the ScsiOpenPath operation has been performed, the SCSI Manager accepts Receive and Send commands from other initiators on the SCSI bus.

Managing Data Packets

Receive and Send commands operate to transmit data packets back and forth between the SCSI Manager processor target and other initiators. It is up to the system services or application programs executing both at the SCSI Manager's system and the other initiator's system to agree on the content and meaning of these data packets beforehand. The SCSI Manager is responsible only for the secure transport of the data packets and makes no attempt to interpret them.

Operating Methods for Target Mode Commands

In target mode, the commands operate in two principal ways. One method requires that a buffer be provided to the SCSI Manager for transmitting or receiving data *before* the Receive or Send command is issued by an initiator. The other method awaits the arrival of a SCSI command from an initiator before a buffer is provided to the SCSI Manager.

In the first method, `ScsiTargetDataTransmit` and `ScsiTargetDataReceive` are used to inform the SCSI Manager of the location and size of the buffer that may be used to satisfy the next Receive or Send command sent by an initiator. When the data has been transmitted to or received by the initiator and the SCSI command is complete, this completes the `ScsiTargetDataTransmit` or `ScsiTargetDataReceive` operation, and a status code and the SCSI status are returned to the caller.

In the second method, when the SCSI Manager is selected by an initiator, it disconnects from the SCSI bus after the command has been received and before any Data In or Data Out phase on the SCSI bus. This leaves the SCSI command suspended. The originating initiator expects the SCSI Manager processor device to eventually reconnect and complete the command by transferring the data requested.

Using a Target Check or Wait Operation

A system service or application may be informed about the arrival of a Receive or Send command by means of the `ScsiTargetCdbCheck` and `ScsiTargetCdbWait` operations. The first operation ascertains whether a command has arrived and returns status code 0 (“ercOK”) if there is a command waiting and status code 387 (“No CDB available”) otherwise. The second waits and does not return to the caller until a command is waiting. Once a command is waiting, it may be completed by providing a buffer to the SCSI Manager in a call to the `ScsiTargetDataTransmit` or `ScsiTargetDataReceive` operation.

Example of Using the SCSI Manager

If you are writing a system service to control a particular class of SCSI devices (optical scanners, for example), the first things the system service needs to know are the number of devices present in the device class and their SCSI addresses. Figure 18-2 shows an example of a code fragment that interrogates the SCSI bus to determine the device class and ready status of all devices attached to the SCSI bus. The logic used in the figure is essentially the same as that used by the file system during hardware initialization to determine whether SCSI disks are present.

In Figure 18-2, neither path parameters nor a password are supplied in the call to `ScsiOpenPath`. “Specifying Path Parameters” states that omitting the path parameters block causes default values to be used for all fields. Because no password is provided, a SCSI path can only be opened for those targets and LUNs for which a SCSI device password has not yet been established or for which no path in exclusive mode has been created.

In practice, this means that the code shown in Figure 18-2 would not be able to establish paths to any SCSI direct-access devices. For such devices, `ScsiOpenPath` fails with status code 219 (“Access denied”) because the file system has already established a SCSI device password. The example code would, therefore, not be able to recognize any such devices on the SCSI bus. Likewise, any other SCSI devices that have already been opened in exclusive mode or have had a password established by another user program would not be detected.

Figure 18-2. SCSI Manager Example

```
{
    char                cdb[12];
    char                inquiryData[36];
    unsigned            cbInquiryData, cScanner, erc, ph;
    unsigned            scannerPh[MAX_SCANNER];
    unsigned short      hostAdapter, targetId, lun,
                        status;
    for (targetId = 0; targetId <= 7; targetId++)
        for (lun = 0; lun <= 7; lun++)
        {
            erc = ScsiOpenPath(pbScsiManagerName,
                               cbScsiManagerName, hostAdapter, targetId, LUN,
                               &ph, NULL, 0, NULL, 0, MODE_PEEK);
            if erc == ercOK
            {
                memset(cdb, '\0', sizeof(cdb));
                cdb[0] = INQUIRY;
                cdb[4] = sizeof(inquiryData);
                erc = ScsiCdbDataIn(ph, 0, cdb, 6,
                                    inquiryData, sizeof(inquiryData),
                                    &cbInquiryData, &status, NULL, 0);
                if (erc == ercOK && status == GOOD)
                {
                    if (inquiryData[0] == SCANNER)
                    {
                        erc = ScsiClosePath(ph);
                        cScanner++;
                        erc = ScsiOpenPath(pbScsiManagerName,
                                           cbScsiManagerName, hostAdapter,
                                           targetId, LUN, &scannerPh[cScanner],
                                           NULL, 0, NULL, 0, MODE_EXCLUSIVE);
                        memset(cdb, '\0', sizeof(cdb));
                        cdb[0] = TEST_UNIT_READY;
                        erc = ScsiCdbDataIn(scannerPh[cScanner],
                                           0, cdb, 6, NULL, 0, NULL, &status, NULL, 0);
                    };
                } else
                {
                    erc = ScsiClosePath(ph);
                };
            }
        }
}
```

Once the path is established to the desired target and LUN, the code shows a SCSI Inquiry command being used to determine the kind of SCSI device, if any, that is present. If no SCSI device with the specified ID is connected and powered on, the `ScsiCdbDataIn` operation fails with a status code that indicates no answer to the selection attempt. If a device is present, the `ScsiCdbDataIn` operation normally completes with a good status code and good target status. In this case, information that describes the kind of SCSI device present, the name of the manufacturer, the model name of the device, and so forth, is returned. The first byte of the inquiry data may be examined to determine the device class (for example, whether it is direct access, sequential access, a printer, a scanner, or some other type).

When a SCSI device of the desired class (in this example, Scanner) is detected, it is good practice to close the initial SCSI path (which was opened in peek mode) and reopen a new path either in exclusive mode or in shared mode with a password. This precaution allows the application or system service gaining control of the device to prevent unauthorized access by other users.

After a SCSI device in the desired class has been detected, a typical system service might perform a Test Unit Ready command to see if the device is fully operational. Because this command does not have a data phase, either `ScsiCdbDataIn` or `ScsiCdbDataOut` may be used to issue the command. When there is no data phase, the pointer to the data buffer, the size of the data buffer, and the parameter for the return of the actual transfer count are all optional.

Each SCSI path that does not lead to a SCSI device of the desired class needs to be closed before attempting an Inquiry command on the next SCSI path. This is necessary to release system resources in the SCSI Manager for reuse by another program.

SCSI Management Operations

The SCSI device management operations described below are grouped according to their function. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

SCSI Paths

GetScsiInfo

Performs the same function as *ScsiQueryInfo*. (See *ScsiQueryInfo*.)

ScsiManagerNameQuery

Returns the name of the SCSI Manager at the specified processor.

ScsiOpenPath

Establishes a path between the calling program and a SCSI target device and LUN connected to a SCSI host adapter. The path handle returned is used to refer to the SCSI target device and LUN in all subsequent operations directed to the device.

ScsiQueryInfo

Returns information about devices connected to a SCSI bus. *GetScsiInfo* performs the same functions but *ScsiQueryInfo* should be used in all new applications.

ScsiQueryPathParameters

Returns a path parameter block that describes the current operational characteristics of a SCSI path.

ScsiSetPathParameters

Allows the caller to modify, by means of a new path parameter block, the current operational characteristics of a SCSI path.

ScsiClosePath

Deletes a previously established path between a program and a SCSI target device and LUN connected to a SCSI host adapter. Any system resources associated with the path are freed for subsequent reuse.

Basic SCSI I/O

ScsiCdbDataIn

Issues a command to a SCSI device and provides for an optional Data In phase for the transfer of data from the device to memory.

ScsiCdbDataOut

Issues a command to a SCSI device and provides for an optional Data Out phase for the transfer of data from memory to the device.

ScsiRequestSense

Issues a Request Sense command to a SCSI device and provides for the transfer of sense data from the device to memory.

Advanced SCSI

ScsiCdbDataInAsync

Issues a command to a SCSI device and provides an optional Data In phase for the transfer of data from the device to memory. Control is returned immediately to the calling program. To verify the completion of the SCSI operation, the program must make a subsequent call to the ScsiWaitCdbAsync operation.

ScsiCdbDataOutAsync

Issues a command to a SCSI device and provides an optional Data Out phase for the transfer of data from memory to the device. Control is returned immediately to the calling program. To verify the completion of the SCSI operation, the program must make a subsequent call to the ScsiWaitCdbAsync operation.

ScsiWaitCdbAsync

Waits for the completion of an asynchronous SCSI operation (either ScsiCdbDataInAsync or ScsiCdbDataOutAsync), checks the status code, and obtains the byte count of data transferred and the final target status.

ScsiReset

Asserts the RST signal on the SCSI bus. This terminates all operations in progress and resets all SCSI devices.

Target Mode

ScsiTargetCdbCheck

Checks if a Send or Receive command descriptor block has been received from another initiator on the SCSI bus.

ScsiTargetDataReceive

Provides a buffer for data received from another initiator on the SCSI bus during the Data Out phase of the Send command issued by the other initiator.

ScsiTargetDataReceiveAsync

Provides a buffer for data received from another initiator on the SCSI bus during the Data Out phase of the Send command issued by the other initiator. Control is returned immediately to the calling program. To verify the completion of the SCSI command, the calling program must make a subsequent call to the `ScsiWaitTargetDataAsync` operation.

ScsiTargetDataTransmit

Provides a buffer for data to be transmitted to another initiator on the SCSI bus during the Data In phase of the Receive command issued by the other initiator.

ScsiTargetDataTransmitAsync

Provides a buffer for data to be transmitted to another initiator on the SCSI bus during the Data In phase of the Receive command issued by the other initiator. Control is returned immediately to the calling program. To verify the completion of the SCSI command, the calling program must make a subsequent call to the `ScsiWaitTargetDataAsync` operation.

ScsiTargetCdbWait

Checks if a Send or Receive command descriptor block has been received from another initiator on the SCSI bus. If no command has arrived, this operation waits.

ScsiWaitTargetDataAsync

Waits for the completion of an asynchronous SCSI target mode operation (either the `ScsiTargetDataReceiveAsync` or `ScsiTargetDataTransmitAsync` operation), checks the status code, and obtains the byte count of data transferred and the final target status.

ScsiTargetOperationsAbort

Cancels any pending `ScsiTargetDataReceive`, `ScsiTargetDataTransmit` or `ScsiTargetCdbWait` operations that are in progress and awaiting selection by another initiator on the SCSI bus.

Section 19

Generic Print Access Method

What is the Generic Print Access Method?

The Generic Print Access Method (GPAM) is a library of object module procedures that send text formatting commands to an output device. GPAM is a high-level, device-independent I/O programmer interface.

You would typically use GPAM if you wish to add a variety of formatting characteristics to text you output to a printing device. GPAM's formatting commands communicate with the output device through the Sequential Access Method (SAM).

(For details on GPAM, see the *Generic Print System Programming Guide*.)

Section 20

Structured File Access Methods

What is Structured File Access?

The file management system provides access to disk file data as randomly addressable, 512 byte sectors. Up to 128 sectors can be read or written in a single request. Data is transferred directly between disk and the buffer specified in the read/write request (that is, it is not buffered by the file system). Asynchronous operation (concurrent I/O and computation on behalf of the same process) is a standard feature of the file management system.

The structured file access methods augment the capabilities of the file management system. The file access methods are object module procedures that can be linked to application programs as required. (For details on linking programs, see the *CTOS Programming Utilities Reference Manual: Building Applications*.) These object module procedures provide buffering and use the asynchronous I/O capabilities of the file management system to automatically overlap I/O and computation.

In contrast to the file management system, which organizes disk file data as unstructured 512 byte sectors, the structured file access methods organize disk file data in one of the following ways:

- As a sequence of variable-length records
- As a sequence of fixed-length records

Files are organized as a contiguous sequence of records. They are both *blocked* (as many records as possible are stored in each physical sector) and *spanned* (logical records are permitted to cross physical sector boundaries).

Generally, a file is created and accessed by one of the following file access methods:

- The Indexed Sequential Access Method (ISAM) or the direct access method (DAM), if the file is a sequence of fixed-length records
- The record sequential access method (RSAM), if the file is a sequence of variable-length records

Note that SAM is an unstructured file access method. SAM is used to create and to subsequently access a file consisting of an unstructured sequence of bytes called a *byte stream*. (See the section entitled “Sequential Access Method,” for details.)

ISAM

ISAM provides both random and sequential, nonoverlapped I/O. *Non-overlapped* means that a call to an ISAM operation does not return to the application program until an associated I/O is complete.

Unlike RSAM and DAM, ISAM allows a file to be accessed by more than one user at the same time. This is possible because a system service intercepts the requests to access files and coordinates performing the desired services on behalf of the requestors.

An ISAM data set consists of two files: an index file and a data file.

The data file contains fixed length data records. The index file provides rapid, ordered access (based on record keys) to the information contained in the data file records.

ISAM allows the user to designate certain fields in each record as keys. For each key field, the index file contains pointers to records in the data file; these pointers are sorted based on the key field values. Say, for example, all records are of the following format:

First name
Last name
Address

The index file could contain record pointers that are sorted alphabetically by the contents of the field, *Last name*.

When an ISAM data set is created or reorganized, the following items are specified:

- The record size
- The number and location of record keys
- The type of each key (integer, real, character, and so forth)
- The method of ordering (alphabetic, numeric, ascending, descending)

ISAM consists of object module procedures in the library, *ISAM.lib*. These procedures send requests to the ISAM system service. ISAM is a separately purchasable software product that includes the system service, the library procedures, and utility programs for maintaining and reporting on ISAM data sets.

For details, see the *CTOS Indexed Sequential Access Method (ISAM II) Programming Reference Manual*.

RSAM

RSAM provides sequential, overlapped I/O. *Overlapped* means that although the application program makes a call to an RSAM operation and that operation returns, I/O can continue concurrently (overlapped) with the computations of the application program.

An RSAM file is accessed as a sequence of variable-length records. Files can be opened for read, write (which replaces any prior file content), and append. In addition to pure sequential access, there are operations for scanning forward to the next well-formed record following detection of a malformed record.

Only one user at a time can access an RSAM file to modify its contents.

RSAM consists of object module procedures in the standard operating system library.

DAM

DAM provides random, nonoverlapped I/O.

A DAM file is accessed as a sequence of numbered, fixed-length records. Random access is by record number; the implementation is such that reading or writing records with sequential record numbers provides good sequential performance. Files can be opened for read or modify. Selective modification is permitted for prior file content.

Only one user at a time can access a DAM file to modify its contents.

DAM consists of object module procedures in the standard operating system library.

Hybrid Access Patterns

In the following paragraphs, the terms ISAM data store file, RSAM file, and DAM file are used to denote the primary means by which the file is accessed.

This usage, while convenient, is oversimplified: any file created with ISAM, RSAM, or DAM can be physically viewed as unstructured and accessed using SAM. Similarly, any file of records created with DAM or ISAM can be physically accessed using RSAM (that is, treating fixed-length records as a special case of variable-length records). Finally, an ISAM data store file contains fixed-length records and therefore can be accessed using DAM.

Although all these hybrid access patterns are possible, they are not all advisable. For example, reading a DAM file with SAM fetches control bytes along with the DAM record bytes; interpreting these requires special knowledge. Also, the file header for ISAM data store files, RSAM files, and DAM files contains a byte used to identify the file type. Accessing the file with a different access method can alter this byte. For example, if an ISAM data store file is accessed with DAM, it is marked as a DAM file and cannot be accessed by ISAM operations unless an ISAM Reorganize is done. [See the *CTOS Indexed Sequential Access Method (ISAM II) Programming Reference Manual* for details.]

An ISAM data store file has an associated index file that must be updated in a complex way when the data store file is modified. If the data store file is modified using ISAM, this is done automatically. If the data store file is updated otherwise, the integrity of the ISAM data set can easily be destroyed. (See the ISAM programming reference manual for details.)

The hybrid access patterns listed below are both useful and safe:

- Use of RSAM or DAM to read an ISAM-created file as though it were an unkeyed DAM file, that is, with the records accessed according to their physical ordering.
- Use of RSAM to read, write, or append to a DAM-created file. (However, if, following a write or append to such a file, there are records of different lengths, the file is subsequently accessible only with RSAM, not with DAM.)
- Use of DAM to read or modify an RSAM-created file in which all records have the same length.

Modifying and Reading Data Files

The **Maintain File** command can modify and/or read RSAM and DAM data files. Maintain File can do all of the following:

- Verify the file structure
- Remove malformed records
- Remove deleted records
- Move valid records to the start of the file
- Optionally write a log of the verification of the file structure to a video display

Maintain File is described in the *CTOS Executive Reference Manual*.

Maintain File also can be used before the **ISAM Reorganize** command. [See the *CTOS Indexed Sequential Access Method (ISAM II) Programming Reference Manual* for details.]

ISAM data store files, RSAM files, and DAM files are standard access method files. As such, they contain standard record headers, record trailers, and file headers.

A *physical record* consists of the record header, the record data, and the record trailer stored in contiguous bytes.

A standard file header is located at the beginning of the first sector at the start of the file. The header consists of information common to all standard access methods followed by information unique to the particular access method. The first physical record is located at the beginning of the second file sector.

The structure of a standard file header, a standard record header, and a standard record trailer are given in the section entitled "System Structures," in the *CTOS Procedural Interface Reference Manual*.

Selecting a File Access Method

Two factors should be considered when selecting a file access method to use: disk space and access time. Each of these factors is discussed below.

Disk Space

RSAM file records take the least amount of disk space because the records are variable length. Each record is only as long as the data it contains so there is no wasted space.

The records in a DAM file usually take up more disk space than RSAM files. Because DAM records are of a fixed length, the record size must be large enough to accommodate the largest block of data. Space is wasted by records containing less data than the largest block.

An ISAM data set takes up the most disk space. Essentially, an ISAM data set is a DAM file (fixed length records) with an extra index file for rapid access to the records. In some cases, the size of the index file can be much greater than that of the data file.

Access Time

ISAM data store files are designed to minimize the time required to read and write random data records. A record can be added anywhere in a file and can be accessed directly using a key rather than by searching the entire file sequentially.

A DAM file is also designed for quick access if the user knows the number (position) of the desired record in the file. Usually, however, this information is not known without an indexing arrangement.

RSAM files take the longest time to access when reading records randomly. To access a given record, every record between the current file pointer and the desired record must be read. For this reason, RSAM files are only suitable when an application reads records sequentially anyway.

Structured File Access Methods Operations

The structured file access methods provide the operation listed below. (See the *CTOS Procedural Interface Reference Manual* for a complete description.)

GetStamFileHeader

Copies the file header of an RSAM, DAM, or ISAM file into the specified area.

Section 21

Indexed Sequential Access Method

What is ISAM?

The Indexed Sequential Access Method (ISAM) provides efficient, yet flexible, random access to fixed-length records identified by multiple keys stored in disk files.

Each ISAM data set holds one type of data record. The size of the data records, the number of keys, and the type of each key are specified when an ISAM data set is created.

For details on ISAM, see the *CTOS Indexed Sequential Access Method (ISAM II) Programming Reference Manual*.

Section 22

Record Sequential Access Method

What is RSAM?

The Record Sequential Access Method (RSAM) provides efficient sequential access to variable-length records. Records are read and written using sequential, overlapped I/O. Records are both blocked (as many records as possible are stored in each physical sector) and spanned (logical records are permitted to cross physical sector boundaries). There is also an operation to scan forward to the next well-formed record following detection of a malformed record. Files can be opened for read, write (which replaces any prior file content), and append.

RSAM can be called directly from any supported programming language. RSAM consists of object module procedures contained in the standard operating system library.

RSAM Files and Records

RSAM provides efficient sequential access to fixed- and variable-length records in a file. An RSAM file is a sequence of these records. Unless all records are of the same length (special case of variable-length records), random access is not feasible.

The data portion of a record can be as large as 65,527 bytes or as small as 1 byte. In addition, a record always contains 7 bytes of structured file access data. To provide efficient disk storage, records are blocked and spanned.

If a sector cannot be read or a record is malformed, the remainder of the file can be read after the ScanToGoodRsRecord operation is used to locate the next well-formed record.

RSAM Working Area

RSAM uses a work area supplied by the application program. A record sequential work area (RSWA) is a 150 byte memory work area for the exclusive use of the RSAM procedures. Multiple RSAM files can be open simultaneously using separate RSWAs.

RSAM Buffer

RSAM also uses a word-aligned buffer supplied by the application program. The buffer must be at least two sectors (1K byte) long. The buffer size is not constrained by the longest record to be read or written. However, in the case that records are long, performance can be improved by using large buffers.

RSAM uses overlapped output. Therefore, completion of the RSAM call `WriteRsRecord` does not mean that the data in the buffer has actually been written to the file. The `CheckpointRsFile` operation flushes the RSAM buffer, ensuring that all data is written to disk.

RSAM Operations

The RSAM operations described below are categorized as basic or advanced. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Basic

OpenRsFile

Opens or creates an RSAM file.

ReadRsRecord

Reads the next record from an RSAM file.

WriteRsRecord

Writes a record to an RSAM file.

CloseRsFile

Closes an RSAM file (including conclusion of all I/O operations).

Advanced

SetRsLfa

Sets the logical file address at which the next I/O operation will occur.

GetRsLfa

Returns the logical file address at which the next I/O operation will occur.

ScanToGoodRsRecord

Scans forward to the next well-formed record in an RSAM file.

CheckpointRsFile

Checkpoints the open output RSAM file.

ReleaseRsFile

Same as *CloseRsFile* but does not write the remaining buffers to disk. This operation is used for abnormal closures.

Section 23

Direct Access Method

What is DAM?

The direct access method (DAM) provides efficient random access to fixed-length records. A record is referred to in DAM by the record number within a file.

DAM can be accessed in COBOL through COBOL Relative I/O. DAM can also be called directly from any of the supported programming languages. DAM consists of object module procedures in the standard operating system library.

In reading, writing, or deleting, DAM does simple address calculations based on the record size and record number to find the required sectors of the DAM file. DAM keeps a cache of recently referenced sectors that are obtained without reference to the disk. Sectors not in the cache are accessed with a single disk access.

DAM Files, Records, and Record Fragments

DAM provides efficient random access to records identified by the record number within a file. The record number of the first record in a DAM file is 1.

A DAM file is a sequence of fixed-length records. The length of a record is specific to each DAM file and is specified when the file is first created.

A record can be as large as 63,992 bytes or as small as 0 bytes. A record also contains the standard 8 bytes of header and trailer in addition to the stored data of the record itself. To provide efficient disk storage use, records are both *blocked* (as many records as possible are stored in each physical sector) and *spanned* (logical records are permitted to cross physical sector boundaries).

A *record fragment* is a contiguous area within a record. A record fragment is specified using an offset from the beginning of the record to the start of the fragment and a byte count that indicates the fragment size. The record fragment must be contained within the record.

Record fragments are read from and written to open DAM files using the operations `ReadDaFragment` and `WriteDaFragment`, respectively.

DAM Working Area

DAM uses a work area supplied by the application system. A *direct access work area* (DAWA) is a 64 byte memory work area for the exclusive use of the DAM procedures. Any number of DAM files can be open simultaneously using separate DAWAs.

DAM Buffer

DAM also uses a word-aligned buffer supplied by the application program. The buffer size is specified by the program. The size is subject only to the constraint that it be a multiple of 512, and that it be greater than or equal to the record size plus 519.

This constraint can be relaxed in two cases:

- If 512 is a multiple of the record size plus 8, the minimal size is simply 512.
- If the record size plus 8 is a multiple of 512, the minimal size is the record size plus 8.

Buffer Size and Sequential Access

DAM reads from and writes to the buffer by using a single request to the file management system. This typically requires only a single disk access. Whenever the disk is read, the entire buffer is filled.

If the buffer size is chosen to be larger than the record size (by at least a factor of 2), the buffer acts as a look-ahead cache. If sequentially numbered records are requested, DAM typically finds them in the buffer and does not access the disk. In this way, if the application program makes a suitable buffer size choice, DAM can provide efficient record sequential access.

Buffer Management Modes

DAM provides two modes of buffer management: write-through and write-behind. The mode is initially set to write-through when a DAM file is opened. The mode can be changed using the `SetDaBufferMode` operation.

In the *write-through mode*, DAM immediately writes the changed sectors of the buffer to disk whenever a record is written or deleted. DAM guarantees that the file content on disk is accurate at the completion of a modify operation.

In the *write-behind mode*, DAM writes changed sectors of the buffer to disk only when new sectors are brought into the buffer, the DAM file is closed, or the mode is changed to write-through. Write-behind mode provides better performance when DAM is used to modify records in sequential order.

DAM Operations

The DAM operations described below are categorized as basic or advanced. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Basic

OpenDaFile

Opens or creates a DAM file.

ReadDaRecord

Reads a record from a DAM file.

WriteDaRecord

Writes a record to a DAM file.

DeleteDaRecord

Deletes a record from a DAM file.

CloseDaFile

Closes a DAM file.

Advanced

QueryDaRecordStatus

Copies to the specified area the status of a record in an open DAM file.

QueryDaLastRecord

Copies to the specified area the number of the last record in an open DAM file.

TruncateDaFile

Truncates an open DAM file (that is, it removes all records beyond a specified point).

ReadDaFragment

Reads a record fragment from an open DAM file.

WriteDaFragment

Writes a record fragment to an open DAM file.

SetDaBufferMode

Sets the buffer management mode to write-through or write-behind.

Section 24

Memory Management

What is Memory Management?

Your application can use the memory management operations described below to obtain memory dynamically for storing code and data.

Memory Management Terminology

Terms relating to memory management are described in Table 24-1. You may need to refer to this table as you read about memory management in this section.

Table 24-1. Memory Management Terms

Term	Definition
16-bit addressing	Using a logical address with a 16-bit offset to address memory. This type of addressing allows access of up to 64K bytes of memory.
32-bit addressing	Using a logical address with a 32-bit offset to address memory. This type of addressing allows access of up to 4 gigabytes of memory.
Expand down segment	A segment in which the offset is greater than the limit.
Expand up segment	A segment in which the offset is less than or equal to the limit.
Heap	Dynamic memory consisting of segments that can be allocated and deallocated in any order.
Huge segment	A segment that can be greater than 64K bytes.

continued

Table 24-1. Memory Management Terms (cont.)

Term	Definition
Logical memory address	A memory address as perceived by an application, consisting of a segment address and offset.
Long-lived	Persisting across chain to another program in the partition.
Offset	The distance, in bytes, of the target location from the beginning of the segment.
Segment	A contiguous memory area in the linear address space.
Segment address	The segment base address in linear memory (real mode) or a selector (protected mode).
Short-lived	Existing only for the duration of current program execution and is deallocated when the program terminates.

Partition Memory

Partition memory is memory an application can allocate dynamically from the partition. It is of two types: short-lived and long-lived.

Short-lived memory exists only for the duration of program execution and is automatically deallocated at program termination. Unless explicitly deallocated by an application, *long-lived* memory persists across a chain to another program.

Within an application partition, short-lived memory expands downward from high memory locations while long-lived memory expands upward from low memory locations.

The operating system allocates short-lived memory for code and static data when an application is first loaded into memory. No explicit use of memory management operations by the programmer is necessary to do this.

To obtain additional short-lived and long-lived memory, your application can make requests of the operating system.

Because long-lived memory is deallocated only at the explicit request of each application, it is useful for passing information from one application to the next in the same partition. Long-lived memory is deallocated, however, when an application is replaced by the Executive.

Global Linear Address Space

On virtual memory operating systems, an application can dynamically obtain memory from the global linear address space. (For details on global linear addresses, see “Virtual Memory Operating Systems,” in the section entitled “Overview of Operating System Concepts.”) The addresses obtained are not associated with the memory allocated to the application partition. Using special memory management calls, an application can allocate up to the maximum amount of linear address space that is available. (See “Using Global Linear Addresses.”)

Segments

A *segment* is a contiguous area in the linear address space. The operating system uses *segmented addressing*. This means every address is relative to a segment. Current compilers and the Linker support segments up to 64K bytes in size.

A *paragraph* is 16 bytes of memory. Segments are aligned on paragraph boundaries.

Addressing a Byte in a Segment

It is conventional to address a byte within a segment by using a logical memory address. A *logical memory address* consists of the following:

- A 16 bit *segment address* (SA)
- A *relative address* (RA) (called an offset)

In real mode, the SA is the segment base address. The *segment base address* is the first byte of the segment in the linear address space.

In protected mode, the SA is a selector (SN). The SN is the index of a segment descriptor entry in either a local descriptor table (LDT) or a global descriptor table (GDT). (For details on protected mode structures, see the Intel manuals listed in “Related Documentation.”)

The segment descriptor contains the segment base address. Protected mode programs should not depend upon the value of SA because there is no correspondence between it and the location of the segment base in memory.

The RA or segment offset of a logical memory address is the distance, in bytes, of the target location from the beginning of the segment.

A byte in memory does not have a unique logical memory address. You can specify the same byte using many different SA and RA combinations.

Segment Limit and Huge Segments

In a protected mode segment descriptor, the segment limit defines the maximum segment size. The processor interprets the limit value based on the granularity specified for it. A 16-bit limit field that is byte granular, for example, defines a 64K byte segment. A *huge segment* is a segment that can be greater than 64K bytes with a maximum size of 4 gigabytes. It has a 20-bit limit field with 4K byte granularity.

Segments of up to 64K bytes can be accessed by programs using 16-bit addressing (logical addresses with 16-bit offsets). To access the memory in huge segments beyond the first 64K bytes, however, requires using 32-bit addressing (32-bit offsets).

Code, Static Data, and Dynamic Data Segments

The three types of segments are code, static data, and dynamic data. Each segment type can be either shared or exclusive.

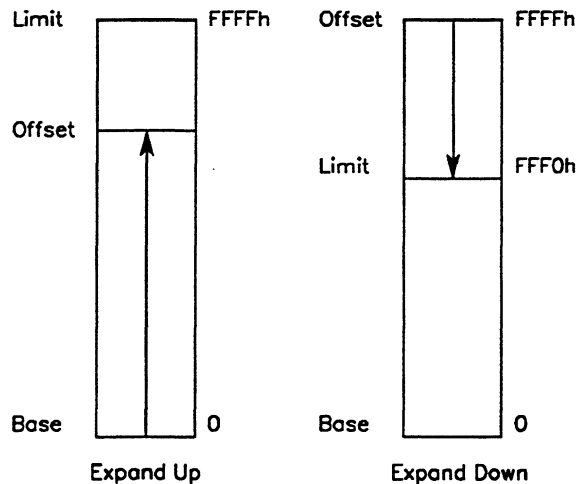
- A code segment contains only processor instructions (code) and, as a rule, is not modified once it is loaded into memory. Such a segment can be shared by several processes. The paging service and the virtual code management facility can read it from the run file as needed without saving a copy of it in memory.
- A data segment contains writable data. There are no restrictions on modifying the contents of a data segment. If a data segment is shared among processes, concurrency control is the responsibility of those processes.

A static data segment is automatically loaded into memory when the run file that contains it is loaded. A dynamic data segment is allocated by a program in memory by means of run-time calls to the operating system.

Expand Up and Expand Down Segments

A data segment selector can have the attribute of being expand down or an expand up. Expand up segments always have a limit address higher than the segment offset address. The reverse is true of expand down segments. (See Figure 24-1.)

Figure 24-1. Expand Up and Expand Down Segments



557.24-1

Expand down segments can be used by the application to readjust the size of a segment already allocated without creating a new selector. (See "Allocating and Deallocating Partition Memory," for details on the operations available to do this.) Expand up segments are created by the operations for allocating global linear address space. (See "Using Global Linear Addresses.")

From Source Modules to Program in Memory

A program on disk is stored in a run file that contains code and/or static data segments. When requested, the operating system loads the program into a memory partition and adjusts any logical memory addresses that exist in either code or data segments to reflect the memory address at which the program is loaded. (See Figure 24-2.)

Code and static data segments are created by compiling and/or assembling source language modules into object modules and linking the object modules together into code and data segments.

The Linker reads the object module(s) and combines them according to their segment names, class names, and directives from the user.

(For details on the Linker, see the *CTOS Programming Utilities Reference Manual: Building Applications*.)

Models of Segmentation

Segments can be combined based on a series of different segmentation models. Although the small and large model are used, most programming languages use the medium model of segmentation. (For details, see your programming language documentation. See the section entitled “Stack Format and Calling Conventions” in *CTOS/Open Programming Practices and Standards* for a discussion of the medium model.)

Run Files

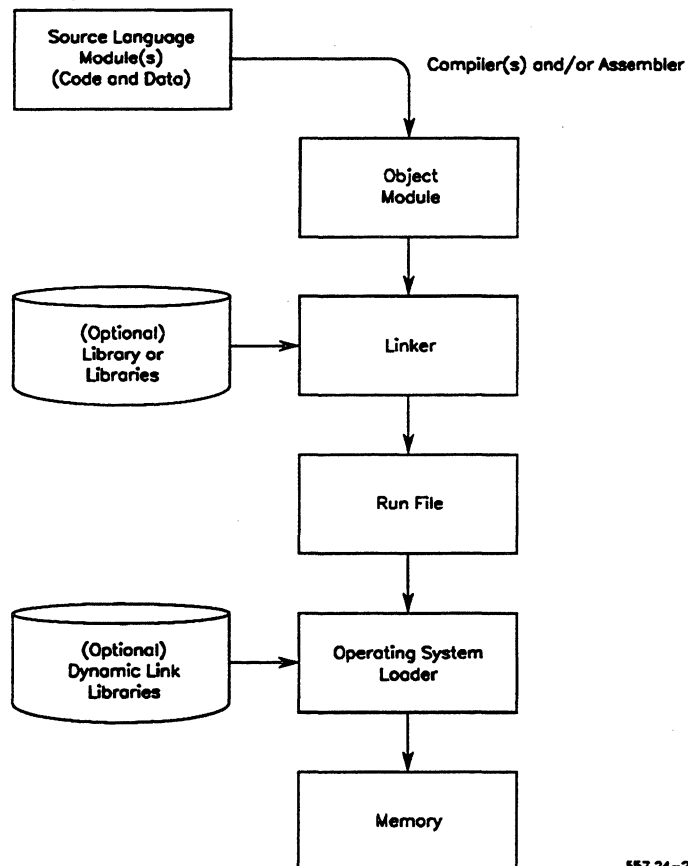
A run file created by linking object modules produced by the Pascal compiler, for example, consists of one code segment for each object module included in the link and a single static data segment. The single static data segment, or DGroup, combines the static data and stack requirements of all the object modules.

A run file of this form is considered standard; assembly language programmers are urged to adopt this standard unless other considerations are overriding. (See “Using Linear Memory.”) The COBOL compiler and BASIC interpreter do not produce object modules. (See your programming documentation for details.)

A program can allocate a dynamic data segment of memory by means of run-time calls to the operating system.

On multipartition and variable partition operating systems, the virtual code management facility allows a program to run that is larger than the available memory in an application partition. When in use, virtual code management loads all the static data segments and the resident code segment into memory but the programmer controls which code segments are placed in overlays that are loaded as needed. (For details, see the section entitled "Virtual Code Management.")

Figure 24-2. From Source Modules to Program in Memory



557.24-2

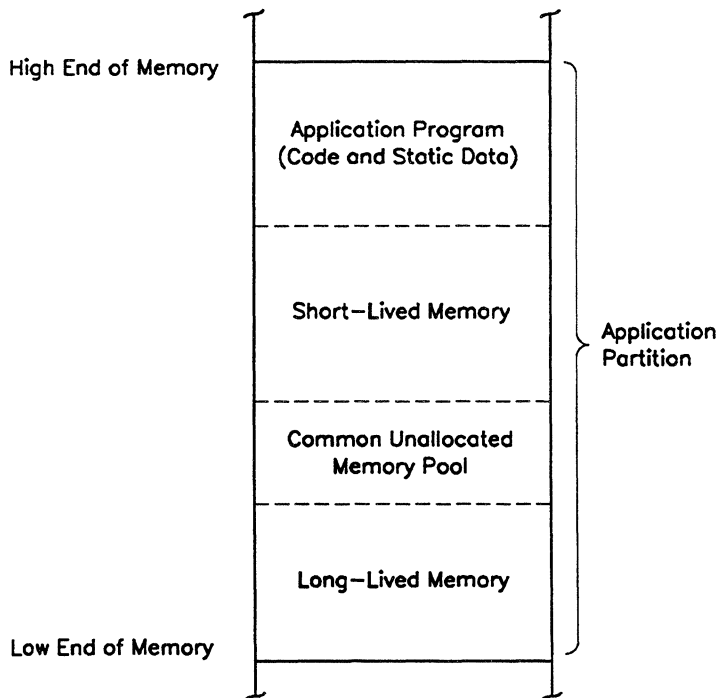
On virtual partition operating systems, the virtual code management facility is not needed. Applications that use overlays, however, can run without change. The overlays are simply treated like ordinary program pages: they are faulted into physical memory frames by the paging service when the code they contain is requested.

Application Partition Memory Organization

Figure 24-3 shows the memory organization of an application partition.

Note: *Figure 24-3 is a logical model of an application partition. For details on the history of partitions, see "Memory Management Styles" in the section entitled "Introduction to Operating System Concepts."*

Figure 24-3. Application Partition Memory Organization



557.24-3

Allocating and Deallocating Partition Memory

All currently unallocated long-lived and short-lived memory in an application partition is in a contiguous area called the common unallocated memory pool. (See “Application Partition Memory Organization.”) Memory can be allocated from both ends of the pool. There is no restriction on how much can be allocated from either end, other than that the sum of the allocations cannot exceed the amount of memory available in an application partition. The `QueryMemAvail` or `QueryBigMemAvail` operation returns the size of all available memory in an application partition.

Memory is allocated and deallocated only on paragraph boundaries. That is, the physical address of the area is a multiple of 16. Because of this, the areas of memory the operating system allocates can be conveniently referenced by using the segment addressing convention discussed in “Segments.”

The `AllocMemoryLL`, `AllocAreaSL`, and `AllocMemorySL` operations allocate long-lived (LL) and short-lived (SL) memory segments in an application partition. The `AllocAllMemorySL` operation can allocate more than 65,536 bytes, and thus the entire area allocated by this operation is not necessarily addressable as a single segment.

The `DeallocMemoryLL` and `DeallocMemorySL` operations deallocate long-lived and short-lived memory segments, respectively, in an application partition. The `ResetMemoryLL` operation deallocates all long-lived memory in an application partition.

The `ExpandAreaLL` and `ExpandAreaSL` operations increase the size of a segment previously allocated using the `AllocMemoryLL` or `AllocAreaSL` operations, respectively. (Segments allocated with `AllocMemorySL` cannot be expanded.) If the Linker’s DS allocation option is specified, `ExpandAreaSL` also may be used to increase the size of the static data segment, DGroup. (See the *CTOS Programming Utilities Reference Manual: Building Applications* for details.)

The `ShrinkAreaLL` and `ShrinkAreaSL` operations decrease the size of a segment previously allocated using the `AllocMemoryLL` or `AllocAreaSL` operations, respectively. (Segments allocated using `AllocMemorySL` cannot be decreased in size.)

For details and examples of how to use the memory management operations for allocating and deallocating memory, see *CTOS/Open Programming Practices and Standards*.

Order of Deallocating Partition Memory

Relative to allocations from one end of the memory of an application partition, deallocations must occur in exactly the opposite sequence. That is, the user must follow a last-allocated, first-deallocated discipline when deallocating either long-lived or short-lived memory. For example, if a program allocates short-lived memory segments A, B, and C, it must deallocate them in the order C, B, A.

Thus the motion of the borders of the common memory pool (dashed lines in Figure 24-3) in an application partition resembles the playing of an accordion: the borders converge when memory is allocated and diverge when memory is deallocated. This scheme is efficient because all unallocated memory is in a common pool and because the operating system has to remember only the addresses of the next (long-lived and short-lived) segments to allocate, not the addresses of all allocated segments.

Using Long-Lived Memory

The long-lived memory in an application partition is used to pass variable length parameter block (VLPB) parameters from one program to a succeeding program in the same partition. A program cannot place 32 bit logical memory addresses in long-lived memory. This is because the long-lived memory of a variable partition can be relocated when a program terminates and is replaced by a succeeding program with different memory requirements. (For details on variable partitions, see “Memory Management Styles” in the section entitled “Introduction to Operating System Concepts.”)

Long-lived memory allocations are returned to the common pool of unallocated memory in an application partition upon explicit request of the program or if the program is replaced by the Executive, which itself calls `ResetMemoryLL` during its initialization.

Using Short-Lived Memory

The short-lived memory in an application partition is used by the operating system to contain the code and static data segments of each application program. If code is shared, however, code can be located anywhere in memory. Short-lived memory also is allocated by application programs for use as dynamic data segments for data that is to be processed only by the current program. Other common uses of short-lived memory are I/O buffers and the Pascal heap.

Short-lived memory allocations are returned to the common memory pool whenever the program is replaced in any application partition by the Chain, ErrorExit, or Exit operations, or by the key combination **Action-Finish**. (See “Program Termination.”)

Using Global Linear Address Space

On virtual memory operating systems, protected mode applications can dynamically allocate and deallocate memory in the global linear address space. Two groups of operations are available to manage the memory in different ways.

To manage huge segments addressed by a single selector, an application can use the operations listed below:

- AllocateSegment
- DeallocateSegment
- ResizeSegment

To manage contiguous regions of memory addressed by a sequence of selectors, an application can use the following operations:

- AllocHugeMemory
- DeallocHugeMemory
- ReallocHugeMemory

The memory obtained can be used to manipulate dynamic data, for example, to manage the pixels in a bit map. It cannot be used to increase the size of the static data segment.

Obtaining Memory Addressed by a Single Selector

The `AllocateSegment` operation allows the application to obtain a huge segment from the available global linear address space. The operation returns a single selector describing the segment. Programs that use `AllocateSegment` must use assembly language routines or a compiler that permits 32-bit addressing to access the memory obtained.

To resize a segment allocated by `AllocateSegment`, an application must call the `ResizeSegment` operation. `DeallocateSegment` can be used to deallocate the segment.

Any program can use `AllocateSegment` to manipulate the memory obtained as a heap. To address the memory, a program using 16-bit addressing simply needs to specify a segment size of 64K bytes or less in each call to allocate a segment. `AllocateSegment` allows the caller to freely deallocate segments in any order. The operations for obtaining partition memory, on the other hand, require deallocation in a specified order. (For details, see “Order of Deallocating Partition Memory.”)

Obtaining Memory Addressed by a Selector Sequence

The `AllocHugeMemory`, `ReallocHugeMemory`, and `DeallocHugeMemory` operations are designed to allow medium and large model 16-bit programs access to large areas of the global linear address space. `AllocHugeMemory` returns the first of a sequence of selectors. Each selector allocated (except the last) addresses 64K bytes; the last selector addresses 64K bytes or less.

`ReallocHugeMemory` can be used to resize the memory allocated up to the maximum size specified, or it can be used to resize any expand up segment up of 64K bytes or less. Resizing is performed at the high address end of the memory allocated by `AllocHugeMemory` or the high end of the expand up segment.

`DeallocHugeMemory` deallocates all the memory allocated by `AllocHugeMemory`. An application cannot selectively deallocate segments.

Memory Management Operations

The memory management operations described below are categorized by use. Operations are arranged alphabetically in each group. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Short-Lived Memory

AllocAllMemorySL

Allocates the largest possible short-lived memory segment in an application partition.

AllocAreaSL

Creates a short-lived segment and allocates memory for it of the specified size. *AllocAreaSL* returns a 32 bit logical address of the base of the segment.

AllocMemoryFramesSL

Creates a short-lived segment and allocates *cFrames**4096 bytes of short-lived memory at the beginning of the segment. The beginning of the segment is aligned on a 4K byte boundary in physical memory.

AllocMemorySL

Similar to *AllocAreaSL*, except that the segment may not subsequently be increased or decreased in size. However, the offset portion of the 32 bit address returned is guaranteed to be 0, enabling the segment to be addressed using only the 16 bit segment base address portion.

DeallocMemorySL

Deallocates a short-lived memory segment in an application partition.

ExpandAreaSL

Allocates additional memory of the specified size within the specified short-lived segment. The specified segment must have been created by a prior call to *AllocAreaSL* (except when specifying the Linker's DS allocation option and *ExpandAreaSL* to expand the static data segment, DGroup).

ShrinkAreaSL

Deallocates memory of the specified size within the specified short-lived segment. The segment must have been created by a prior call to *AllocAreaSL* (except when specifying the Linker's DS allocation option and *ShrinkAreaSL* to decrease the size of the static data segment, DGroup).

Long-Lived Memory

AllocMemoryLL

Allocates a long-lived memory segment in an application partition.

DeallocMemoryLL

Deallocates a long-lived memory segment in an application partition.

ExpandAreaLL

Allocates additional memory of the specified size within the specified long-lived segment.

ResetMemoryLL

Deallocates all long-lived memory in an application partition.

ShrinkAreaLL

Deallocates memory of the specified size within the specified long-lived segment.

Short-Lived and Long-Lived Memory

DefineInterLevelStack

Initializes the stack segment (SS) and the stack pointer (SP) fields of the caller's task state segment for the specified protection level. *DefineInterLevelStack* is supported in protected mode only.

DefineLocalPageMap

Defines an address mapping between the linear address specified by *saLocal* and the physical address referenced by *pFrames*. This operation is supported on 80386 and higher microprocessor-based operating systems.

QueryBigMemAvail

Returns the size in bytes of all available free memory in an application partition. It is recommended that applications use this call rather than *QueryMemAvail* (described next) since *QueryMemAvail* can only return a value up to 1 megabyte.

QueryMemAvail

Returns the size in paragraphs of all available free memory in an application partition.

ShrinkPartition

Reduces the size of the application partition to the amount of memory currently allocated to short-lived memory.

I/O Management

QueryIOOwner

Returns a record structure describing one or more processes with exclusive access rights to a specified range of I/O addresses. Each process description is contained in a 10-byte record.

ResizeIOMap

Changes the size and the linear memory address of the 80386 I/O permission bit map located in the caller's Task State Segment (TSS).

SetIOOwner

Establishes or releases exclusive access rights to one or more contiguous ranges of I/O addresses specified by the caller.

Selector Management

CreateAlias

Returns an alias selector for the specified source memory address. The alias selector combined with a 0 offset references the same linear address as the specified source memory address.

DeallocSg

Deallocates a global descriptor table (GDT) selector (SG).

DeallocAliasForServer

Asserts that the alias selector is no longer in use and may be deallocated by the operating system.

FValidPbCb

Verifies that the data area specified is accessible by the calling program.

GetSegmentLength

Accepts a memory address and returns the length of the segment.

LaFromP

Returns the 32 bit linear address pointed to by the logical memory address *P*. (If your application needs physical addresses to program the hardware, it should use the DMA buffer management operations. See “Using DMA Buffers” in the section entitled “Bus Address Management,” for details.)

LaFromSn

Returns the linear address (not *physical address*) referenced by the protected mode selector *Sn*. *LaFromSn* is supported by operating systems executing in protected mode only and is invoked by *LaFromP*.

RemakeAliasForServer

Remakes the alias selector so the selector will persist across system-common return and Respond.

ReuseAlias

Rewrites the base address and limit fields of an alias selector initially allocated by *CreateAlias*. *ReuseAlias* is supported in protected mode only.

ReuseAliasLarge

Is the same as *ReuseAlias* except that it provides a more general interface. *ReuseAliasLarge* is supported in protected mode only.

SetSegmentAccess

Sets the access mode of a code or a data segment in protected mode. *SetSegmentAccess* performs no function in real mode.

SgFromSa

Returns an alias GDT selector (SG) that references the same memory location as the specified SA.

SnFromSr

Returns the protected mode SN that references the same memory location as the specified real mode segment address (SR). SnFromSr is supported only by operating systems executing in protected mode.

SrFromSn

Returns the real mode SR that references the same memory location as the specified protected mode selector (SN). SrFromSn is supported only by operating systems executing in protected mode.

Global Linear Address Space Management

AllocateSegment

Creates a short-lived segment in the linear address space of the specified size.

AllocHugeMemory

Allocates memory of the specified size and allocates a sequence of expand up selectors to address it.

DeallocateSegment

Deallocates the segment specified by the selector.

DeallocHugeMemory

Deallocates memory allocated by AllocHugeMemory.

ReallocHugeMemory

Resizes either the memory allocated by AllocHugeMemory or an expand up segment up to 64K bytes owned by the caller.

ResizeSegment

Changes the size of the specified segment previously created by AllocateSegment.

Section 25

Cacher

What is the Cacher?

This section describes the operating system cacher. The cacher operations are system-common procedures that allow a program to temporarily store and maintain fixed-sized blocks of data in memory in the anticipation that the data will be requested again. To determine which operating system versions support use of the cacher, see Appendix A, “Operating System Features.”

A cacher usually incorporates a mechanism for updating the data stored in memory to reflect what is currently being requested. To accomplish this, the operating system cacher uses a least-recently-used (LRU) algorithm. Although this algorithm identifies data to be removed when space is needed, the procedures for moving data into and out of the cache are the responsibility of the calling program. The LRU algorithm, combined with a hashing technique for locating requested data in a small subset of what is actually cached, are key features that increase performance of the cacher as well as the programs that use it. Other services provided by the cacher are searching for requested cache entries, determining whether data is currently in the cache, and reporting statistics on cache performance.

This section points out the advantages of the cacher service and describes how to use the cache operations in programs you write. You are also provided an overview of cacher operation to assist you in optimizing the use of this facility.

Cacher Terminology

The terms in Table 25-1 are used to describe features of the operating system cacher.

Table 25-1. Cacher Terms

Term	Definition
Writethrough	Describes a cache that keeps secondary storage synchronized with the data in the cache. At the completion of each request made to modify data, a write-through cache immediately writes the data back to disk.
Writeback	Describes a cache that can contain data not matching what is in secondary storage.
Dirty	Describes a write-back cache entry with data that has been modified but has not yet been written to secondary storage.
Sticky	Describes a cache entry that is to remain in the cache (is not subject to removal from the cache according to the LRU algorithm).
Locked	Describes a cache entry whose buffer is currently being used. Locked entries are taken off the LRU list until they are released by the user with a call to <code>CacheReleaseEntry</code> .
Valid	Describes a cache entry that has a unique cache identification number (cache ID).
Stealable	Describes an available cache entry on the LRU list.
Cache hit	Indicates that the requested data buffer is in the cache.
Cache miss	Indicates that the requested data buffer is not in the cache, but an alternate (stealable) buffer is available.

Why Use the Cacher?

The file management service uses the cacher operations to cache disk file data. File management's use of these operations, however, is only one application. The cacher is a generic set of operations that can be used to enhance the performance of many programs. It can benefit your program in the following ways:

- It increases performance by maintaining the most currently referenced data in memory. This reduces the overall time necessary to access requested data. Data handling provided by the cacher is more flexible than that provided by the memory disk device, for example, which is always memory resident. (See "Memory Disk" in the section entitled "Disk Management.")
- It services both read and write operations.
- It caches different types of data. Using the cacher operations allows your program the flexibility of caching any type of data, not just a select group of disk files. File management uses the cache operations for the express purpose of caching disk sectors. Using the cache operations, your program can cache X-Blocks, communications packets, complex structures, or whatever data you elect to cache.
- Optionally, it can retain data in memory at the request of the caller. For example, your program can construct a detailed, frequently used structure once and store that structure in the cache by asserting the sticky status for that entry. When the structure is no longer needed, it can be removed by calling a cache operation to change its status. This saves the time required to reconstruct the structure each time it is needed.
- Although buffer size is set at cache pool initialization, the cacher system-common procedures allow the caller to obtain a group of buffers, which the caller can treat as a single, logical buffer.

- It allows the programmer to create as many caches as needed to perform different functions. Each cache created using the `CacheInit` operation is assigned a unique cache pool handle. The handle is used to distinguish the cache from all others that may exist in subsequent operations. The cache operations can be used by single applications or by system services.
- It maximizes the use of available memory. A program allocates the memory for a cache through calls it makes to the operating system. Depending on the hardware, up to 64 megabytes of memory can be allocated for use by the cacher.

Cache Data Structures

A cache consists of two types of data structures: a single cache pool descriptor and, for each cache entry, a cache entry descriptor. The structures are used to format the memory allocated for a cache when the `CacheInit` operation is called. “Initializing a Cache” describes how these structures appear in memory.

Cache Pool Descriptor

There is one cache pool descriptor for each cache created. The cache pool descriptor is the root structure of the cache data structures. It contains the memory addresses of doubly linked lists threading the cache entries together.

Entries are threaded together in two ways:

- One set of head and tail pointers maintained in the cache pool descriptor links entries containing free (available) data buffers in LRU order. This queue of available buffers or *LRU list* contains two types of entries: *invalid* entries for which cache identification numbers (IDs) have not yet been defined and *valid* entries with defined IDs. (Cache IDs are described in “Cache Entry Descriptor.”)
- An entry is also linked to one of several *hash queues*. The head and tail of each hash queue are maintained in an array in the cache pool descriptor. How an entry is hashed to one of these queues is described in “Hashing.”

(For the format of the cache pool descriptor, see “Cache Pool Descriptor” in “System Structures” in the *CTOS Procedural Interface Reference Manual*.)

Cache Entry Descriptor

For each entry in a cache, there is a cache entry descriptor. This structure points to the cache data buffer, describes the current state of the entry (such as whether the buffer is sticky or dirty), and indicates the entry’s link position in the LRU list and in a hash queue.

There is also an 8 byte cache ID that uniquely identifies the cache entry. The contents of a cache ID are under the control of the caller. The ID of a disk cache, for example, might contain the virtual disk address, the index of the device control block (DCB), and (on a shared resource processor) the board number for the cache entry. The cache ID is assigned to an entry the first time the CacheGetEntry operation is called to obtain that entry. (See “Obtaining a Cache Entry.”)

(For the format of the cache entry descriptor, see “Cache Entry Descriptor” in “System Structures” in the *CTOS Procedural Interface Reference Manual*.)

Initializing a Cache

To initialize a cache, memory must be set aside for it. Based on the amount of allocated memory and the data buffer size specified by the caller, the number of cache pool entries can be estimated and the cache memory is formatted. Each of these procedures is described in more detail below.

Allocating Cache Memory

Before a cache is initialized, the memory for it must be allocated. To allocate memory, your program can use a memory management operation such as AllocateMemorySL. (For details on obtaining memory, see the section entitled “Memory Management.”)

The cache pool is not necessarily one contiguous area in memory. Each call to allocate memory for the pool allocates a new selector. A selector, by nature, can refer to any segment in memory.

Calculating Number of Cache Entries

Once memory for the cache is obtained, the CacheInit operation is called to initialize the cache. Based on the amount of memory allocated and the data buffer size specified by the caller, CacheInit calculates the number of cache entries in the cache pool using the following formula:

$$(sCacheArea - sCachePoolDesc)/(descriptorSize + sCacheEntry)$$

where

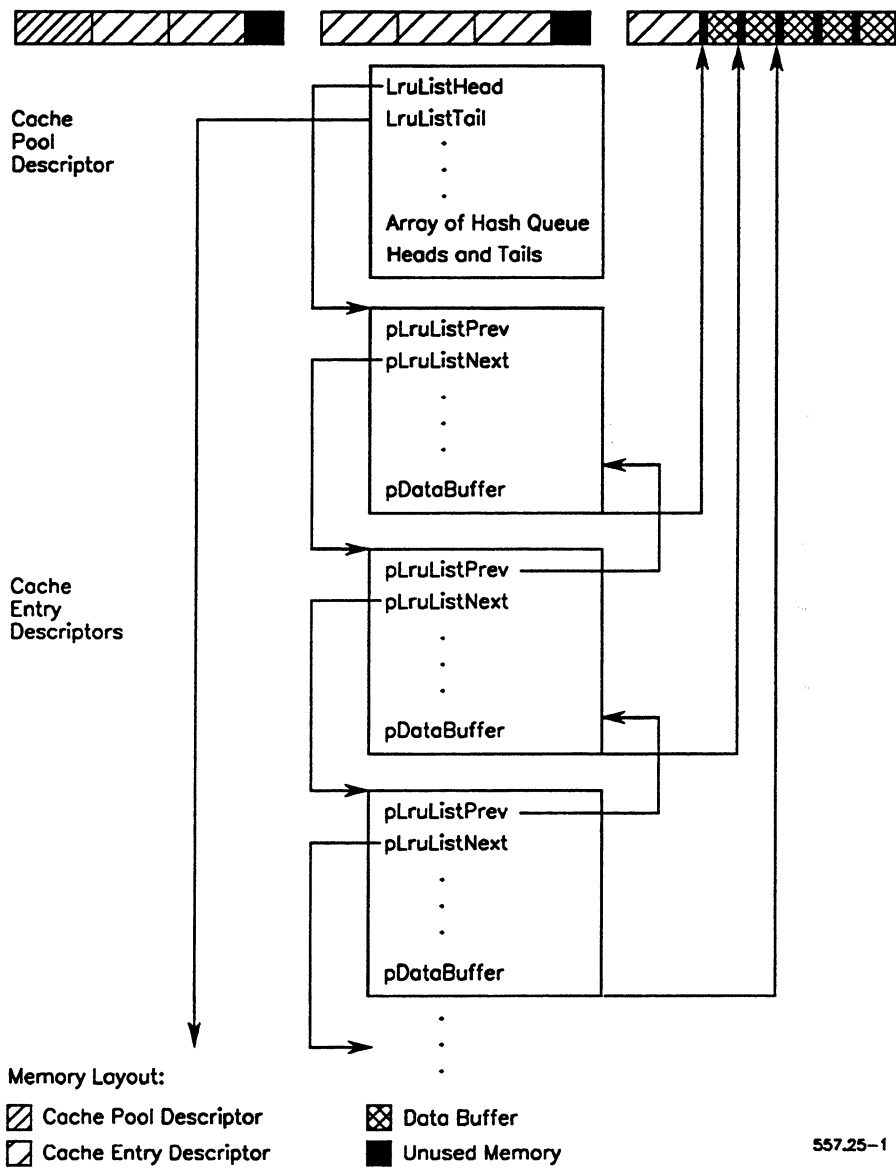
<i>sCacheArea</i>	is the total amount of memory allocated to create the cache pool
<i>sCachePoolDesc</i>	is the size of the cache pool descriptor
<i>descriptorSize</i>	is the size of a cache entry descriptor
<i>sCacheEntry</i>	is the size of the cache entry data buffers

Formatting Cache Memory

The number of buffers calculated by the preceding formula is only an approximation of the actual number created by CacheInit. To format the memory area(s) allocated for the cache, CacheInit uses the following procedure (see Figure 25-1):

1. Reserves memory in first memory segment allocated for the cache pool descriptor. Figure 25-1 shows a simple cache formatted in three memory segments (located towards the top of the figure). The segment containing the cache pool descriptor is shown on the left.
2. Formats the rest of the allocated memory as follows:
 - a. Constructs the cache entry descriptors.
 - b. Divides up the remaining memory for the data buffers. By formatting memory in this manner, the actual number of buffers can be less than the number of cache entry descriptors.

Figure 25-1. Formatting Cache Memory



557.25-1

This can occur, for example, if a block does not fit in an allocated segment or if the data buffers require boundary alignment [because the direct memory access (DMA) service will be used to fill them]. In these cases, some of the memory is not used. Figure 25-1 shows six cache entry descriptors but only five data buffers because of alignment requirements. (The black areas represent memory not used for the cache structures.) If memory is a limited resource, your program can use this memory for other applications.

3. Threads the structures together, as described below:
 - a. Threads the data buffer addresses (one in each cache entry descriptor) to the actual buffers in memory.
 - b. Links each cache entry descriptor with the previous and next in the LRU list. The head and tail of the LRU list are maintained in the cache pool descriptor.
4. Returns a cache pool handle that is used to specify this cache pool in calls to all the other cache operations.

Hashing

The first time that the `CacheGetEntry` operation is called to obtain a buffer associated with an entry, the cache entry is assigned an 8 byte cache ID. This ID, together with the cache pool handle, are provided in subsequent calls to `CacheGetEntry` or `CacheGetStatus` to query the entry. To optimize retrieval of the entry from the cache pool, a hashing algorithm is applied to the ID. The algorithm hashes the entry to one of the several smaller search lists. Each time the cache is queried, the entry is hashed to the same queue and that queue is searched in a linear fashion.

The memory addresses of the heads and tails of all the hash queues are maintained in an array in the cache pool descriptor.

Obtaining a Cache Entry

When the caller needs to perform I/O, it first calls `CacheGetEntry` to see if the desired information to be read or written is already in the cache. To `CacheGetEntry` the caller supplies the cache pool handle returned by `CacheInit` and the memory address of the cache ID.

`CacheGetEntry` is perhaps the most complex of the cache operations. Depending upon the entry's state, different operations are performed by `CacheGetEntry` or by the caller. For example, the following situations can occur:

- The cache entry buffer is found in the cache. This is called a cache hit. `CacheGetEntry` locks the buffer for the exclusive use of the caller. At the return from `CacheGetEntry`, the caller can empty or fill the buffer using an appropriate mechanism, such as DMA.
- The cache entry buffer is not in the cache but there is an alternate buffer available. This is called a cache miss. In this case, the first available buffer in the LRU list is locked out for use by the caller. Upon the return from `CacheGetEntry`, the caller needs to fill the buffer with the appropriate data from secondary storage. The intent in transferring the data to the cache buffer is to make it more likely that the data will be fetched from cache memory (the next time it is needed) rather than from secondary storage.
- The cache entry buffer is currently locked out to a different caller, or the buffer was not found and there are no alternate buffers available. In either of these cases, the caller is responsible for coordinating process scheduling.

Depending on the situation encountered, `CacheGetEntry` returns a word of status information, indicating the appropriate action the caller should take. In addition, `CacheGetEntry` returns a cache entry handle for use in a call to `CacheReleaseEntry` to release the entry after the caller is done emptying or filling the buffer.

Cache Optimizations

Through flag bit settings to `CacheGetEntry`, your program can chain buffers together, defer cache hits, and reserve entries on the LRU list. The latter two features allow the caller some lookahead into the cache to optimize its use.

Chaining Cache Buffers

By setting the flag bit *mCacheChained*, cache buffers can be chained together. In effect, this flag allows your program to vary the buffer size dynamically. If, for example, you wanted to load a program segment from the disk while maintaining minimal disk arm movement, you would ideally like to have a buffer large enough to contain 64K bytes of data.

If *mCacheChained* is set and a call to `CacheGetEntry` results in a cache miss, the first available buffer on the LRU list is locked and becomes the head of a linked list of buffers. If *mCacheChained* is set in a subsequent call to `CacheGetEntry` resulting in a cache miss, the next cache entry buffer obtained would be linked to the head. Each successive call resulting in a cache miss with the flag set would link another buffer on the end of the chain.

This can be done until `CacheGetEntry` results in a cache hit, at which time the I/O can be performed.

Deferring a Cache Hit

To avoid getting a cache hit, your program can set the flag bit *mCacheDeferHit*. Deferring a cache hit places the entry at the end of the LRU list instead of returning it to the caller to be filled right away.

Preventing Buffer Stealing

The flag bit *mCacheDontSteal* can be set to prevent the caller from stealing a buffer should a cache miss occur. In such a case, the order of entries in the LRU list is not disturbed.

Releasing a Cache Entry

After the caller is done using a buffer, the caller must release it so it can be used by other clients. To release a buffer, the `CacheReleaseEntry` operation is called with the cache pool handle returned by `CacheInit` and the cache entry handle returned by `CacheGetEntry`.

Conditions Before Release of an Entry

Before an entry is released, the caller has the option of changing the entry's status. This is accomplished by setting (asserting) or clearing (deasserting) bits in the control mask and control word parameters to `CacheReleaseEntry`. Currently the following bits are operational:

- The dirty bit. If the cache entry is dirty (the data in the buffer has been modified but not written to disk), the caller must provide a flushing procedure to flush the buffer before it can be released. (The format of this procedure is provided in the description of `CacheInit` in the *CTOS Procedural Interface Reference Manual*.) The dirty status only applies to a write-back (asynchronous) cache. A synchronous (write-through) cache, on the other hand, always writes data to disk as soon as the data is modified.
- The sticky bit. An entry can also be sticky. Such an entry is not subject to the LRU algorithm used to remove entries from the cache. A sticky entry remains in a cache until this status is deasserted.
- The valid bit. The caller can invalidate an entry if, for example, an I/O to secondary storage failed.
- The most-recently-used (MRU) bit. If asserted (the default), the entry is placed at the end of the LRU list after being unlocked. Otherwise, the entry is the next to be replaced by a call to `CacheGetEntry` when a cache miss occurs.

At Release of an Entry

Once the caller has specified the entry's status using the control word and mask, the entry is unlocked and is placed at the end of the LRU list (the default unless the MRU bit is FALSE or the entry is to be invalidated). The contents of the entry buffer, however, are still considered valid (can be obtained as a cache hit by the last program that filled the buffer) until the buffer reaches the head of the LRU list and CacheGetEntry is called with a cache miss.

Flushing Cache Entries

Using the CacheFlush operation, any subset of unlocked cache entries in a cache pool can be selected for flushing and having cache entry status changed. Say, for example, the cache is a disk cache and a device just went offline. In this situation, the caller can flush all the entries for that device by providing CacheFlush with a template that reflects the number of the device that went offline. If the entries are unlocked, the same control word and mask provided to the CacheReleaseEntry operation can be manipulated to change the status of the entries. (See "Releasing a Cache Entry.") CacheFlush would search for all entries with IDs matching the device number and change the status to reflect asserted bits in the control mask and word. The buffers of these entries would then be flushed.

An additional flag bit in the CacheFlush control word and mask allows the caller to flush a locked entry.

Obtaining Cache Status

At any time the caller can obtain the status of a specified cache entry by calling the CacheGetStatus operation and providing the cache entry ID. CacheGetStatus returns the status of the cache entry at the specified address. The information returned includes whether the entry is valid, sticky, dirty, or chained.

Obtaining Cache Statistics

At any time the caller can obtain the statistics on a specified cache pool by calling the `CacheGetStatistics` operation. Using this operation, the programmer can adjust parameters that initialize the cache (such as the size of the buffers used) to optimize its performance. The information is returned in the format of a cache statistics block and includes such data as the number of

- Cache entries that are valid, sticky, or dirty
- Free entries on the LRU list
- Calls to `CacheGetEntry`
- Cache hits, cache misses, and times the caller had to wait for a cache buffer

For details on the format of the cache statistics block, see the description of `CacheGetStatistics` in the *CTOS Procedural Interface Reference Manual*.

Closing a Cache Pool

When a cache pool is no longer needed, the cache can be closed and the cache pool handle invalidated by calling the `CacheClose` operation. Before this operation is called, however, the caller must have performed the following operations on each entry in the cache:

- Flushed the buffer
- Cleared the dirty status
- Released the entry

After `CacheClose` is called, the caller must use an operation such as `DeallocMemorySL` to return the cache pool memory space back to the operating system.

Cache Operations

The cache operations are described in alphabetical order below. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

CacheClose

Closes the specified cache pool and invalidates the cache pool handle.

CacheFlush

Permits any subset of unlocked cache entries to be flushed and to have their status changed.

CacheGetEntry

Queries a cache pool for a specified cache entry.

CacheGetStatistics

Returns statistics for the specified cache pool.

CacheGetStatus

Returns status information for the specified entry in the specified cache pool.

CacheInit

Creates a cache pool by formatting the memory allocated by the caller for the cache pool descriptor, the cache entry descriptors, and the cache data buffers.

CacheReleaseEntry

Releases a cache entry associated with the specified cache entry handle in the cache pool and places the entry at the end of the LRU list. *CacheReleaseEntry* also allows the caller to change certain status conditions of the entry.

Section 26

Utility Operations

What are Utility Operations?

The utility operations are standard operating system library procedures that can increase the efficiency of writing applications.

In this section, the utility operations are presented alphabetically by the task they allow an application to perform.

Groups of utility operations are provided for performing the following tasks:

- Accessing resources in disk files
- Accessing resources in memory
- Comparing strings
- Customizing the user interface
- Directing data to a byte stream
- Handling commands
- Managing names
- Manipulating error messages
- Obtaining the system date and time
- Parsing configuration files
- Using workstation hardware IDs

In addition, individual utility operations are provided for performing miscellaneous tasks including

- Building a single-line text editor
- Comparing logical addresses
- Copying files

- Determining monitor resolution
- Informing the user of waiting mail
- Writing records to the system log file

Accessing Resources in Disk Files

Resource management is a set of operations that allow an application to access and modify resources in disk files. Using the resource management operations, an application can get the information necessary to read a resource, add or delete resources, or copy them from one file to another. Because the operations are library routines, they can be used by applications running on operating systems prior to CTOS III.

Note: *The resource operations can be used with version 8 run files and version 6 run files created by the 12.0 or later Linker.*

A separate operation allows applications to access resources in run files that are in memory. (For details, see “Accessing Resources in Memory.”) This functionality is only available to applications running on CTOS III and later operating systems.

What Are Resources?

On CTOS, the term *resource* historically has meant anything an application might need to carry out its normal execution. Memory, for example, is a typical resource an application requests from the memory manager system service.

In the context of resource management, a resource is still a run file requirement but it is more specific. It denotes a read-only block of data bound to an executable run file. As an example, font data included in a run file for printing documents is a resource.

Presentation Manager (PM) applications typically contain icons, strings, and bit maps. These resources are used to display screen data such as windows, dialog boxes, and graphics images. PM resources must be bound to the run file with a PM tool such as the Resource Compiler. Once in the run file, however, these resources can be manipulated using the resource management operations.

A CTOS *data file* (that is, any human readable text file or machine readable binary file) also can be a resource. Symbol files, Code View data, keyboard data blocks, and message files are just a few examples. The contents of these files can be bound to a run file using the resource management operations.

Note: *A dynamic link library (DLL) is a run file type. As such it can contain resources. (For details on DLLs, see the section entitled “Dynamic Link Libraries.”)*

A group of resources in a run file is called a *resource set*. A *binary resource file* is a special output file of the Resource Compiler containing one or more resources. Resource management treats the contents of a binary resource file as a resource set. A CTOS data file can be added to the resource set in a run file using resource management operations.

Why Resource Management?

There is always the possibility that a separate data file or two might be forgotten when transferring an application from one location to another. What's even more perplexing is when an application appears to have all its data files available but the application run file still doesn't work. This can happen if the version of the data files and the run file do not match. File mismatches can occur in the course of moving around several separate files.

Resource management prevents these types of problems by providing the capability of keeping all the data associated with a run file together as one unit.

Because the resource management operations are library routines, they are ideally suited for use in utilities to be run on different operating system versions. They are, for example, the operations driving the Resource Librarian utility. The Resource Librarian provides a command interface for manipulating the resources in run files, binary resource files, and data files. (For details on the Resource Librarian, see the manual entitled *CTOS Programming Utilities Reference: Building Applications*.)

Resource Work Area

Resource management uses an application-provided buffer in memory to oversee and manage changes the application makes to the resources in a file. This buffer is called the resource work area (RWA). The application doesn't need to know the details of the RWA contents other than one table, the resource descriptor table (RDT).

The RDT is a directory to the resources in the file. It accompanies the resource set in the run file when resources are initially inserted.

Each RDT entry is a *resource descriptor*. A resource descriptor contains the following information about a resource:

- Its size
- Its logical file address (lfa) in the file
- Its type code
- Its resource ID

For details on the format of a resource descriptor entry, see "Resource Descriptor" in the *CTOS Procedural Interface Reference Manual* section entitled "System Structures."

The *type code* of an entry is a value identifying a class of resources, such as dialog boxes. (For a list of the resource types supported, see the *CTOS Procedural Interface Reference Manual*.) The class determines the characteristics of its members. The *resource ID* is a value identifying an instance of a resource type in a class. Resource management sorts RDT entries numerically by type code and resource ID and maintains resources in this sorted order in a run file.

Note: *To reduce the frequency of disk accesses, no data is actually copied to a target run file until access to the target run file resource set is ended.*

Most of the time resource management operations update the RDT entries in the RWA, not the actual contents of the resource set in a run file. Only when an application calls the `RsrcEndSetAccess` operation for a specified run file (to end access to the resource set and save the changes to disk) or the `RsrcEndSession` operation (to end the resource session) do the resources actually get written out to the appropriate run files. If access to a *source run file* resource set (that is, a run file containing resources to be copied elsewhere) is ended, the resources are written to a temporary file. They are written to the final destination or *target run file* either when the target file resource set access or the resource session itself is ended. At the end of a resource session, any *outstanding* resource sets (that is, resource sets with modifications not yet written to the target run file) are automatically written to the appropriate run files by the `RsrcEndSession` operation.

Stepping On Files

To add resources to a run file, your application needs to create a new file to which the resources and the rest of the run file contents are copied. (See “Process Overview” for details.) An application should not attempt to add resources to an existing run file. Doing so would result in overwriting resources already in the run file.

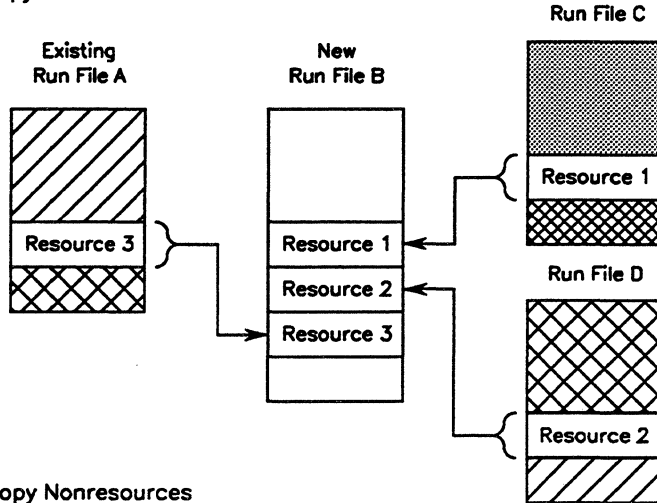
As you recall, resource management maintains run file resources in numerical order by resource type code and ID. It does not, however, adjust the addresses of existing data to accomodate new data added before the end of a run file. It is the responsibility of the application to handle *-old* and *-new* files.

Process Overview

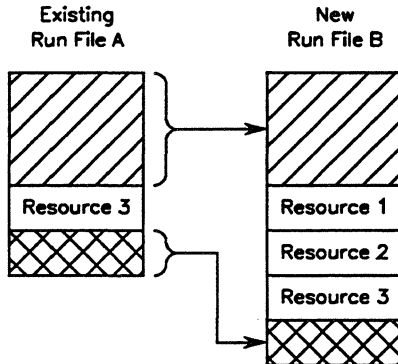
To add resources to an existing file, your application basically creates a new (empty) file. Then it adds the resources and the nonresource portions of the run file to it in separate operations. Figure 26-1 is a simplified overview illustrating how this process can work.

Figure 26-1. Adding Resources to a Run File

(A) Copy Resources



(B) Copy Nonresources



557.26-1

In the figure, existing Run File A contains Resource 3. The procedure outlined below uses resource management operations to add Resource 1 and Resource 2 from Run File C and D, respectively, to Run File B:

1. Create a new empty Run File B.
2. To Run File B, add the specified resources from run files A, C, and D.
3. Copy the nonresource portions of Run File A to Run File B.

The figure is overly simplified but is meant to illustrate that the copy procedure is basically a two-phase process. It actually can be performed in the reverse order. The nonresource portion of the run file can be copied to the new file before the resources.

For each resource session, your application must perform the following functions in the sequence outlined below:

1. Initialize the resource session once for all files with the `RsrcSessionInit` operation.
2. Initialize resource access with the `RsrcInitSetAccess` operation for each run file or binary resource file (but not a CTOS data file). (For details on CTOS data files, see “Copying a Resource from a CTOS Data File.”)
3. Modify the contents of the target file resource set using the resource operations for adding and/or deleting resources.
4. End the resource session with `RsrcSessionEnd`.

In addition, your application should periodically save the interim contents of the target file resources using `RsrcEndSetAccess`. (For details, see “Saving to Disk.”)

Note: *To reduce the frequency of disk accesses, no data is actually copied to the target run file until access to the target run file resource set is ended.*

Starting And Ending a Resource Session

To start a resource session, an application calls the `RsrcSessionInit` operation. `RsrcSessionInit` opens a temporary file on `[Scr]<$>`. The caller must provide an I/O buffer for use by subsequent resource management operations.

To use the resource operations, you must have configured a scratch volume. (See “Optimizing Use of Disk Space” in the *CTOS System Administration Guide* for details on the scratch volume.) If the scratch volume is password protected, the password must be provided to the `RsrcSessionInit` operation.

Resource management only uses the scratch volume if access to a source file resource set is closed with the `RsrcEndSetAccess` operation. (See “Saving to Disk.”) The application doesn’t need to be concerned about this action other than to be aware that a temporary file may be needed to hold data.

At the end of a resource session, the application calls the `RsrcSessionEnd` operation. `RsrcSessionEnd` deallocates the I/O buffer, closes the temporary file on the scratch volume, and writes out the contents of any outstanding resource sets.

Initializing Resource Access

After calling `RsrcInitSession` to start the resource session, your application must call the `RsrcInitSetAccess` operation to set up an RWA for each source and target file involved in a resource data transfer. Resource management uses the RWA to hold data on a resource set and the file in which it is located. Resource management maintains an array of all open RWAs to oversee their status.

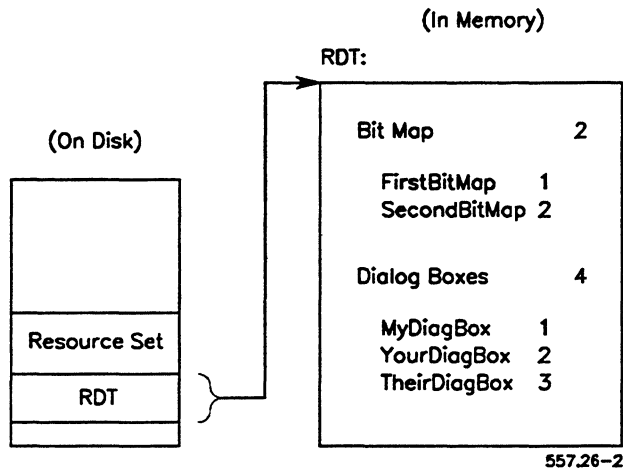
Note: *`RsrcInitSetAccess` is only used with run files or binary resource files. For details on how to handle CTOS data files, see “Copying a Resource From a CTOS Data File.”*

Initializing access to a target run file that doesn’t contain any existing resources simply creates an empty RWA, which resource management fills with resource descriptor information.

Like the memory address of the byte stream work area (BSWA) provided to the sequential access method (SAM) operations, the memory address of the RWA acts as the handle to a resource set. It is provided as the first parameter to the resource access operations to identify the file being worked on and the resource set it contains.

The resource descriptor table (RDT) in the RWA contains an entry for each resource in the run file resource set. Figure 26-2 shows an RDT with five resource descriptor entries of two types: bit map data and dialog boxes. The bit map descriptors are listed first because the bit map type code is the lesser value. Within each resource type, descriptors are arranged from lowest to highest resource ID.

Figure 26-2. Resource Descriptor Table



RsrcInitSetAccess is only used with run files (with or without existing resources) or binary resource files. Resource management knows how to set up the RWA for these file types. (Because the resources in a binary resource file can be in any order, resource management numerically sorts the resource descriptor entries in the RWA for this file type.)

Data files cannot be provided to the **RsrcInitSetAccess** operation. Resource management does not set up RWAs for them. To copy resources from a data file, the application must build a resource descriptor in memory and use the **RsrcCopyFromFile** operation to copy the resource descriptor information to the target run file. (For details, see “Copying a Resource From a CTOS Data File.”)

Saving to Disk

If your application makes frequent modifications to a resource set over a long time period without ending access with the **RsrcEndSetAccess** operation, you run the risk of losing your work. Until such time as **RsrcEndSetAccess** is called, resource set modifications are only made to the target file RDT in memory.

In this situation, we recommend your application periodically call `RsrcEndSetAccess` followed by a call to `RsrcInitSetAccess` again to establish a new RWA. Doing so forces changes made to the target run file resource set to be written to the actual run file. The call to `RsrcInitSetAccess` establishes a new RWA for the next group of resource set modifications.

Frequent calls to end and reinitialize access to the resource set being modified reduces the likelihood of data loss if a power outage or other machine malfunction occurs. This is particularly important if your application is designed with an interactive user interface. Activities such as displaying icon lists, pointing to and clicking on various resources and moving them around to different screen locations all require extensive manipulation of memory addresses. Losing this type of data could halt further execution of your application if provision is not made to safeguard the data with frequent disk writes.

Caution

If there is a machine failure before a call to `RsrcEndSetAccess`, all RWA modifications to the target file resource set are lost.

When `RsrcEndSetAccess` is called, resource management anticipates that your application might plan to close the run file and deallocate the RWA memory. In the likelihood a source file contains resources yet to be written to a target file, `RsrcEndSetAccess` uses the source file RWA as a guide to write the contents of the resource set to the temporary file on the scratch volume.

If your application calls `RsrcEndSetAccess` on the target file, the target file RWA is used as a guide to write out the resources from the source file(s) to the target file. In this situation, the resources can be obtained from either or both of the following locations: the source file(s) and the temporary file, if resource set access has already ended for one or more source file(s).

As stated previously, a call to `RsrcEndSession` completes the task of writing out any resources not explicitly written out to the target file with the `RsrcEndSetAccess` operation. To avoid having to write resources to the temporary file, `RsrcEndSession` operates only on the target files. Then it deallocates the temporary file.

Copying Resources

There are several ways to add resources from source run files to a target run file. The following are some of the possibilities:

- Copy one resource at a time.
- Use an RDT from an existing file as the basis for the resource set in a new target file because it already contains most of the resources you want in a new file; then make minor modifications to the RDT.
- Copy some or all of the resources of a given type in a resource set.
- Copy some or all of the resources in a resource set.

You can also copy a data file resource to a target run file. Using this method, you must provide resource management with resource descriptors in which you have filled in the relevant data yourself. (See “Copying a Resource From a CTOS Data File.”)

Copying One Resource at a Time

To add a single resource from a source run file to a target, you can use the `RsrcCopyFromRsrcSet` operation. To this operation, you provide the resource type code and ID and the addresses of the source file and target file RWAs. Resource management constructs the resource descriptor for the specified resource for your application.

Using a Base RWA

If you just want to add one or two resources to a run file that already contains (let's say) 30 existing resources, you don't have to start over by adding one resource at a time to a new target run file you've created. Instead, your application can call `RsrcInitSetAccess` on the source run file containing the 30 resources. Then, it can create a new empty target run file.

In a second call to `RsrcInitSetAccess` for this newly created file, your application can provide the address of the source file RWA for the parameter `pRwaBase`. This causes resource management to create an RWA containing the 30 resource descriptors from the source file as the starting RDT for the target run file.

The procedure for setting up the base RWA is summarized below (see the description of `RsrcInitSetAccess` in the *CTOS Procedural Interface Reference Manual*):

1. Open an existing file as the source file.
2. Allocate memory for the RWA of the source file.
3. Call `RsrcInitSetAccess`. In this call, provide the value 0 for the parameter `pRwaBase` and the memory address of the source file RWA for `pRwaBuf`, for example
`RsrcInitSetAccess(pFhSourceFile, fileType, 0, pRwaSourceFile, ...);`
4. Create a new (empty) file as the target file.
5. Allocate memory for the RWA of the target file.
6. Call `RsrcInitSetAccess`. In this call, provide the address of the memory allocated for the source file RWA for `pRwaBase` and the memory address of the target file RWA for `pRwaBuf`, for example,
`RsrcInitSetAccess(pFhTargetFile, fileType, pRwaSourceFile, pRwaTargetFile,...);`

Copying Multiple Resources

Two operations allow your application to calculate the number of resources in a file: `RsrcGetCountAllSetType` and `RsrcGetCountSetType`. These operations can be used in conjunction with `RsrcGetAllSetTypeInfo` and `RsrcGetSetTypeInfo`, respectively, to copy multiple resources.

`RsrcGetCountAllSetType` returns the total number of resources in a resource set. This information allows you to calculate the size of the buffer you want to provide to `RsrcGetAllSetTypeInfo` to obtain a copy of the RDT for the entire resource set in a run file.

RsrcGetAllSetTypeInfo uses the RWA of the specified file to obtain the type code and ID of each resource it contains. Then it calls **RsrcGetDesc** repeatedly, providing the type code and ID of each resource. For each type and ID, **RsrcGetDesc** returns the resource information in the format of a resource descriptor in the caller-provided buffer.

Having obtained the buffer of resource descriptors, the application can use the **RsrcCopyFromFile** operation to copy the descriptors, one at a time, to the target run file.

The operations **RsrcGetCountSetType** and **RsrcGetSetTypeInfo** can be used in the same manner as **RsrcGetCountAllSetType** and **RsrcGetAllSetTypeInfo** to obtain all the descriptors of a given type in the specified file.

Copying a Resource From a CTOS Data File

The **RsrcInitSetAccess** operation does not establish an RWA for a CTOS data file because it does not handle this file type. To copy the contents of such a file into a target run file as a resource, your application must build a resource descriptor and fill it in with the pertinent information about the data file. The filled-in descriptor can then be provided as the buffer to the **RsrcCopyFromFile** operation. **RsrcCopyFromFile** copies the resource descriptor to the specified target run file.

Deleting Resources

Deleting resources is accomplished one resource at a time using the **RsrcDelete** operation. Along with the resource handle, the caller provides the type code and resource ID of resource it wants deleted. Like saving a resource to disk, deleting the resource actually takes place when either **RsrcEndSetAccess** or **RsrcSessionEnd** is called.

Copying The Nonresource Portion of a Run File

At any time before or after updating resources in the RDT, the application can use the `RsrcCopyRestRunFile` operation to copy the regions of the run file preceding and following the resource set to the target run file. As necessary, `RsrcCopyRestRunFile` updates the `lfas` of these run file regions to accommodate resources added to or deleted from the resource set. The run file contents are written to disk with either of the save operations, `RsrcEndSetAccess` or `RsrcSessionEnd`.

`RsrcCopyRestRunFile` actually treats the run file regions as if they were resources. The addresses of the regions are maintained in the target run file RWA along with the resource modification data. As such, if `RsrcEndSetAccess` is called on the source run file, the contents of these run file regions written to the temporary file on the scratch volume.

Accessing Resources in Memory

The `GetModuleResourcInfo` operation can be used by an application to access resources in run files executing in memory. It is only supported by applications that are version 6 or version 8 run files executing on a virtual memory operating system.

The caller provides the resource type code and ID. The selector describes a huge segment. Using it, an application can access resources greater than 64K bytes. For details on huge segments, see “Segment Limit and Huge Segments” in the section entitled “Memory Management.”

Comparing Strings

String comparing operations inform you of string equalities. String comparing operations with names starting with the prefix *Nls* can handle nationalized strings.

(For details on nationalization, see the section entitled “Native Language Support.” In addition to describing all the NLS operations, the section describes Extended Native Language Support operations. The ENLS operations can process strings containing single byte and/or multibyte characters.)

Using StringsEqual and NlsULCmpB

StringsEqual states whether two strings contain exactly the same data. StringsEqual does no translation.

NlsULCmpB compares two strings, using uppercase and lowercase translations. Unlike StringsEqual, NlsULCmpB can take nationalized character sets into account. The Executive program uses this operation for interpreting the field entries in a command form or for file matching.

Using WildCardMatch

WildCardMatch checks a string against a wild card specification. The operation returns TRUE if the string matches the specification. (For details on how to use wild cards, see the *CTOS Executive Reference Manual*.)

Handling Nationalized Strings

Other string comparing operations that handle nationalized characters are

- NlsYesOrNo
- NlsYesNoOrBlank
- NlsGetYesNoStrings
- NlsGetYesNoStringSize

NlsYesOrNo uses uppercase and lowercase translations to compare a string against nationalized words meaning *yes* and *no*. The string passed can match any portion of a yes or no word. For example, the strings Y, YE, and YES match YES.

NlsYesNoOrBlank performs the same function as NlsYesOrNo, except that NlsYesNoOrBlank, in addition, checks for a null string.

NlsGetYesNoStrings returns the strings defined in the NLS Yes or No Strings table, and NlsGetYesNoStringSize returns the sizes of these strings.

It is recommended that the NLS operations just described be used in conjunction with the RgParam operation for parsing answers to yes/no options of Executive parameters.

Customizing the User Interface

The following user interface operations are available for customizing the user interface:

EnlsFieldEdit
EnlsFieldEditByChar
FormEdit
QueryZoomBoxPosition
MenuEdit
UnzoomBox
ZoomBox

Using these operations, your program can do any of the following:

- Edit text strings displayed to the video device based on keyboard input from the user or parameter input.
- Create forms in which the user can enter string responses or select from a group of responses.
- Create menus from which the user can select one or more choices.
- Expand and collapse boxes for special video effects or to provide additional online help for given fields in the form or menu.

All the above functions can be performed without linking with a separate, space-consuming forms library. Furthermore, EnlsFieldEdit, EnlsFieldEditByChar, FormEdit, and MenuEdit are nationalizable operations that are especially useful where portability is an issue. (For details on nationalization, see the section entitled “Native Language Support.”)

Directing Data to a Byte Stream

Output routines allow your program to direct information to any byte stream (including the video device) in a way that is compatible with the operating system. The default output device is *[VID]0* (video frame 0).

These operations are replacements for language run time operations. They provide a convenient and efficient way of coding strings in a language such as PL/M, which has no run-time support for displaying strings.

NPrint
OutputBytesWithWrap
OutputQuad
OutputWord
PutByte
PutChar
PutPointer
PutQuad
PutWord
SbPrint
ZPrint

PrintFileClose
PrintFileOpen
PrintFileStatus

All of the output operations use **NPrint** and **PutChar** for output. **NPrint** uses byte streams (described in the section entitled “Sequential Access Method”). You can optionally write your own version of **NPrint** to direct output to a different device(s).

If you intend to send data to a second device concurrently with what is being sent to the video, you can use the three **PrintFile** operations listed above. Using **PrintFile** operations frees you from having to create your own **NPrint** and **PutChar** routines.

PrintFileOpen opens the specified output byte stream in append mode. After calling **PrintFileOpen**, your program can call **PrintFileStatus** to turn on a second output byte stream. With the second output byte stream turned on, the output routines send output to the specified byte stream as well as to the video. **PrintFileStatus** can also be called to query or set the status of the second output byte stream. **PrintFileClose** closes the byte stream.

Handling Commands

The `InitSysCmds` and `GetSysCmdInfo` operations allow you to access commands and obtain information about them. `InitSysCmds` opens a file of commands for the subsequent retrieval of command information. `GetSysCmdInfo` is called to obtain information about a given command, such as the run file, command case, and the number of command parameters. `CloseSysCmds` closes the command file and releases the buffers allocated by `InitSysCmds`.

Managing Names

Name management provides a set of operations to associate names with 32-bit tag values. In a call to name management, the application supplies the name and a tag value it wants associated with it. Name management stores the name in memory in such a manner that the application can quickly and conveniently look up the tag without having to devise its own algorithm to do so.

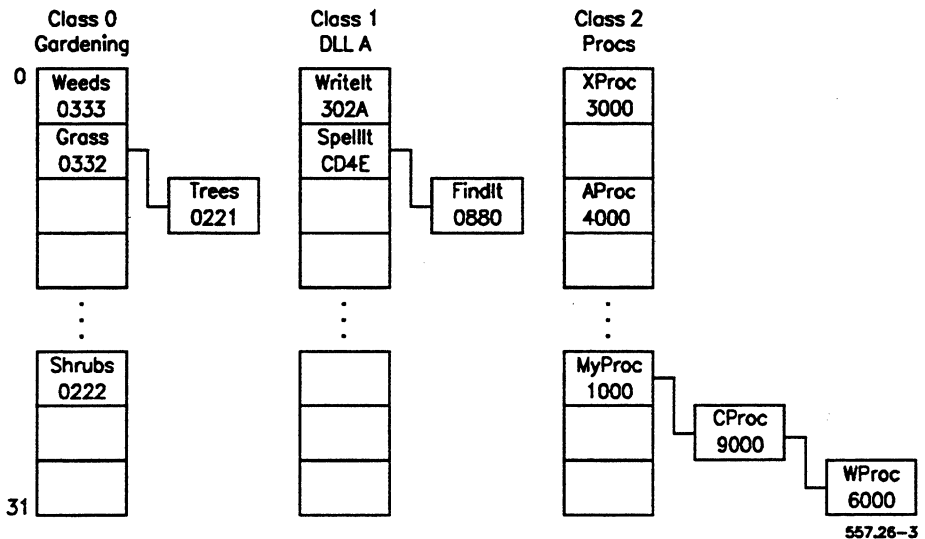
One client of name management is the operating system loader. It stores the addresses of dynamic link library (DLL) procedures as tags that it associates with the procedure names. (For details on DLLs, see the section entitled “Dynamic Link Libraries.”)

Classes

Name management stores names in classes. A *class* is a private name table in which names must be unique. The arrangement of names in classes allows an application to create names in one class without any regard for whether the name is already in another class. As long as each class contains unique names, there are no conflicts.

Collectively, all classes are referred to as the *name heap*. (See Figure 26-3.) Each name table entry in a class consists of a name and its associated tag. When an application requests to register an entry in a specified class, name management applies a hashing algorithm to the class name to determine the relative position of the entry in the class. Sometimes, this means a name and tag hash to a position already occupied by another name and tag. Each extra entry hashing to the same table position is queued to the first in a linked list, as shown in the figure.

Figure 26-3. Name Heap



Configuring the Name Heap

The number of name table positions in a class is a system configurable parameter as is the total size of the name heap. For details on configuration parameters, see the *CTOS System Administration Guide*. Increasing the number of table positions optimizes performance. It reduces the number of extra entries linked to each table position. As a result, less time is spent by name management sequentially searching the linked lists for entries. Conversely, fewer name table positions reduces performance but optimizes memory use by the name heap.

Using the Name Management Operations

Alphabetically, the name management operations are

NameAllocClass
 NameQuery
 NameRegister
 NameRemove

To use name management, an application first needs to allocate a class. This is done by calling `NameAllocClass`. The operation returns a unique 16-bit class identifier (ID). By providing the class ID in a call to the `NameRegister` operation, the application can enter a name and the tag value it wants associated with it into the class. Thereafter, whenever the application needs to retrieve the tag it assigned to a name, it simply calls the `NameQuery` operation, specifying the name and the class it is in. If the application chooses to remove a name from a class, it can do so by calling the `NameRemove` operation. Names and name classes are automatically removed upon the death of the application that registered the name or allocated the class.

Manipulating Error Messages

The following operations allow your program to manipulate error messages:

- `CloseErcFile`
- `GetErc`
- `GetErcLength`
- `GetFileErc`
- `InitErcFile`
- `PrintErc`

Because the error messages are contained in the separate binary error message file *[Sys]<Sys>ErcMsg.bin* (distributed with Standard Software), your program can use this file to retrieve error messages at any time without disturbing your program's normal message file.

`InitErcFile` opens the error message file and allocates the memory for the message work area (MWA). Like the byte stream work area (BSWA) used by the sequential access method (see "Byte Stream" in the section entitled "Sequential Access Method"), the MWA is used internally by `InitErcFile` to maintain message records.

To determine the amount of memory needed to store a message in memory, your program can call the `GetErcLength` operation. `GetErcLength` must be called individually to obtain the length of each message specified.

Once the error message file is open, the `GetErc` or `GetFileErc` operation can be called to retrieve the text of error messages. Depending on the operation selected, the messages can be placed in memory with or without special formatting.

Using the `PrintErc` operation, a message can be placed in a user-supplied byte stream. The `CloseErcFile` operation closes the error message file

Obtaining the System Date and Time

System Date/Time Structure

The System date/time structure represents the system date and time to greater precision than 1 second. If a program executing on a server or standalone workstation needs to know the time to this precision, it can access the structure by calling the `GetPStructure` operation with a structure code of 240.

(`GetPStructure` is described in detail in “Operations,” in the *CTOS Procedural Interface Reference Manual*. For a description of the fields in the system date/time structure, see “System Date/Time Structure” in “System Structures,” in that same manual.)

System Date/Time Format

The system date/time format provides a compact representation of the system date and time to an accuracy of 1 second. It consists of 2 one-word fields in the system date/time structure, namely the *seconds* and *dayTimes2* fields. The system date/time format precludes invalid dates and allows simple subtraction to compute the interval between two dates.

In the high-order word (*dayTimes2*), the high-order 15 bits contain the count of 12-hour periods since March 1, 1952. This 15-bit field allows dates up to the year 2042 to be represented. The low-order bit of *dayTimes2* is 0 to represent AM and 1 to represent PM. The low-order word (*seconds*) contains the count of seconds since midnight/noon. Valid values are 0 to 43199.

The current system date and time are maintained in the server (for all the workstations of a cluster configuration) or in the standalone workstation.

You can access and modify the current system date/time by calling the *GetDateTime* and *SetDateTime* operations.

Expanded Date/Time Format

The expanded date/time format represents the year, month, day of month, and so forth, as discrete fields. The *ExpandDateTime* and *CompactDateTime* operations can be used to convert between the system date/time format and the expanded date/time format. Using the *NlsParseTime* or *ParseTime* operation, the caller can convert a string containing the date and time to the expanded date/time format.

(For details on the fields in the expanded date/time format, see “Expanded Date/Time Format” in “System Structures,” in the *CTOS Procedural Interface Reference Manual*.)

Nationalizing Date/Time Formats

For ease in nationalization, the following native language support (NLS) operations can be used to display the date and time format according to the requirements of a particular language definition:

- NlsStdFormatDateTime*
- NlsParseTime*
- NlsFormatDateTime*

(For details on NLS, see the section entitled “Native Language Support.”)

Parsing Configuration Files

The configuration file operations are used to parse configuration files. A standard configuration file contains entries of the form

:field name: value

Examples of these files include the system configuration file, user configuration files, and Context Manager configuration files. (The *CTOS System Administration Guide* illustrates a user configuration file, which is a typical example.)

The configuration file operations allow your program to parse the following configuration file types:

- User configuration files only
- Any standard configuration file
- Nonstandard configuration files

Parsing User Configuration Files

The following operations can be used to parse standard user configuration files only:

GetUserFileEntry
SetUserFileEntry

These operations conveniently allow the caller to read or modify the contents of a user file without first having to access the Application System Control Block to obtain the user name. (See the *CTOS Procedural Interface Reference Manual* for the format of the Application System Control Block.) The user file suffix (*.user* by convention) is obtained from the nationalization tables.

GetUserFileEntry searches for a field name string entry and returns the value following the trailing colon. SetUserFileEntry searches for a field name string to which it appends a new value following the trailing colon. Both operations start their search from the beginning of the file. If the string is not found, SetUserFileEntry appends it to the file. Both operations open and close the user file for the caller.

These operations cannot be used in conjunction with any of the other configuration file operations.

Parsing Standard Configuration Files

The following configuration file operations can be used to parse any standard configuration file:

- LookUpField
- LookUpNumber
- LookUpReset
- LookUpString
- LookUpValue
- OpenUserFile
- ReadToNextField
- SetField
- SetFieldNumber

Opening and closing the configuration file must be performed by separate operations. It is recommended that only one file be opened and processed at a time.

Caution

Care must be taken when using the standard configuration file operations to open and process multiple configuration files at the same time. When switching to another file, the application must call LookUpReset before performing any operation in the file, or unpredictable results may occur.

To open a configuration file the caller can use either the OpenUserFile operation or OpenFile. OpenUserFile is more specialized in that it can only be used to open a user file, it saves a copy of the file if the file is opened in modify mode, and file suffixes (such as the conventional suffix *-old* for the saved file) are nationalizable.

The operations, however, can only interpret standard configuration file entries using a colon (:) to delimit field names from field values.

The “lookup” operations return information about standard configuration file entries such as the field string length, the value following the trailing colon, or the contents of the entire entry. LookUpString, for example, searches for a field name string entry from the current file position and returns the value following the trailing colon.

The “set field” operations allow the caller to modify the contents of the field string or the value. `SetField`, for example, starts at the current file position and searches for a field name string to which it appends a new value following the trailing colon. `ReadToNextField` reads everything in the file up to the specified field name.

Parsing Nonstandard Configuration Files

The following operations can be used to parse nonstandard as well as standard configuration files:

- `ConfigOpenFile`
- `ConfigCloseFile`
- `ConfigGetNextToken`
- `ConfigGetPosition`
- `ConfigGetRestOfLine`
- `ConfigSetFilePosition`

The operations are used separately from the other configuration file operations. (See “Parsing User Configuration Files” and “Parsing Standard Configuration Files.”) `ConfigOpenFile` and `ConfigCloseFile` must be used to open and close the configuration files, respectively. File contents can be read but not modified, and only one file can be opened at a time.

The operations, however, provide greater parsing flexibility than the operations for parsing standard files. Using the `ConfigGetNextToken` operation, for example, the caller can retrieve a string delimited by uniquely specified starting and ending delimiter characters. The delimiter can be a symbol such as the slash (/) used in some UNIX-based files.

Furthermore `ConfigGetNextToken` can read subfield values individually or all at once, or it can read a value associated with the next configuration file entry. How `ConfigGetNextToken` reads the file is controlled using the flag *fNewLine* and the starting and ending delimiters.

When *fNewLine* is TRUE, ConfigGetNextToken will return the string enclosed in the specified delimiters on the next line if the first delimiter of the specified delimiter pair immediately follows the carriage return. Otherwise (if any other characters precede the starting delimiter on the new line) ConfigGetNextToken ignores the remainder of the line and searches for the delimited string on the next new line.

Say, for example, ConfigGetNextToken is called to read the configuration file contents shown below. In the call to ConfigGetNextToken, the colon (:) is specified as the value of the starting and the ending delimiter parameters, and *fNewLine* is set to TRUE.

```
:Apples:           12 13
                  :14:
:Oranges:          15
```

The first call to ConfigGetNextToken returns the string, Apples. When called a second time with the same parameters, ConfigGetNextToken returns the string, Oranges. The tab preceding :14: on line 2 causes ConfigGetNextToken to ignore the rest of line 2 and to look for the delimited string on line 3. If *fNewLine* is FALSE, the second call would return the string, 14. ConfigGetRestOfLine can be used to return the remaining contents of a line after a field has been located.

To include a delimiter character within a field string, surround it with matching quotation marks. The quotation marks will not be returned as part of the field. The following are examples:

Field	Value	Returns	Delimiter Included
:AField:	"YZ"	YZ	'
:AnotherField:	'Y'Z'	Y"Z	"
:AThirdField:	Y""Z	Y"Z	"

The ConfigGetPosition operation can be used to save the starting point of a group of information in a file. The operation returns the logical file address where the next input operation should start. ConfigSetPosition can then be called to resume reading the file at the specified point.

Using Unique Workstation Hardware IDs

Workstations support an intelligent hardware identification number (ID) device that is connected to the I-Bus between the CPU and the keyboard. The hardware ID device contains nonvolatile memory that can be programmed to uniquely identify a workstation of a given hardware type. Up to 126 unique hardware IDs are supported.

To set the hardware ID of a workstation, an application uses the `WriteHardId` operation. The ID can be read by the `ReadHardId` operation.

When the hardware ID device is being read from or written to, the keyboard and all other devices down stream of the keyboard are disconnected from the CPU. During this time, all input from those devices is lost and no outbound communication is possible. Therefore, it is recommended that the hardware ID be read only during system initialization by an application that saves the ID for other applications.

The hardware ID device and its interfaces allow a system initialization file to be created for an individual workstation rather than for all the workstations of a given hardware type.

Performing Miscellaneous Tasks

In addition to the groups of utility operations described in this section, there are individual operations for performing miscellaneous functions. In the paragraphs that follow, these operations are presented by the task they perform.

Building a Single-Line Text Editor

The `TextEdit` operation is useful for building a single-line text editor. You can call the `TextEdit` operation if you want to write a program that allows the user to do either of the following:

- Enter data into a field.
- Edit the data entered by means of the normal keystrokes of `BACKSPACE`, `LEFT ARROW`, `RIGHT ARROW`, and `CODE+LEFT ARROW`, such as those used by the Executive.

Comparing Logical Addresses

FComparePointer compares two logical addresses for equality. **FComparePointer** typically is used to compare the binary values of the logical addresses. However, it also can be used to compare the byte locations of the addresses in the linear memory address space. (For details on memory addresses, see “Addressing Memory” in the section entitled “Using CTOS Operations.”)

Copying Files

The **CopyFile** operation creates a copy of the current contents of an open file. It is commonly used to save a copy of the current version of a file before the file is again modified. (The saved file is called a *-old* file because the special suffix *-old* is appended to the file name. See “File Naming Rules” in the *CTOS Executive User's Guide* for details.)

Because **CopyFile** does not use byte streams, it can be used by any of the structured file access methods, ISAM, SAM, or DAM (described in “Structured File Access Methods.”) Furthermore, this operation conserves memory that would otherwise be needed to link byte streams with the calling program.

Determining Monitor Resolution

The **GenResString** operation returns a string that describes the resolution of the caller's monitor, such as 1024X768 for a VM-003 monitor with a GC-003 graphics controller.

Writing Records to the System Log File

The **WriteLog** operation can be used to write variable length records to the system log file. (For the format of the Log File, see “Log File Record Format” in the section entitled “System Structures” in the *CTOS Procedural Interface Reference Manual*. Writing to the Log File is described in the *CTOS Programming Guide*.)

Informing User of Waiting Mail

The **QueryMail** operation can be used by any program to inform the user that new electronic mail is waiting. The Executive, for example, uses this operation to display the mail message on the video status line.

Utility Operations

The utility operations described below are categorized by use. Operations are arranged alphabetically in each group. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Accessing Resources in Disk Files

RsrcCopyFromFile

Copies the resource descriptor from a source file to the specified target run file RWA.

RsrcCopyFromRsrcSet

Copies the resource descriptor with the given type and ID from the specified source file RWA to the specified target file RWA.

RsrcCopyRestRunfile

Copies information on the location and size of the source run file contents (resources excepted) to the target run file RWA.

RsrcDelete

Deletes the resource descriptor with the given type code and ID from the specified resource set.

RsrcEndSetAccess

Copies the resource set associated with this RWA to a temporary file for any resource descriptors in this RWA copied to other RWAs. If any resource descriptors from other RWAs are copied to this RWA, all the resources in this RWA are copied to the resource set in the associated run file on disk. If, in addition, *RsrcCopyRestRunFile* has been called on this RWA, the nonresource parts of the run file also are copied to the run file.

RsrcGetSetTypeInfo

Returns the descriptor of every resource in the specified resource set in the buffer provided.

RsrcGetCountAllSetType

Returns the total number of resources in the specified resource set.

RsrcGetCountSetType

RsrcGetCountSetType returns the number of resources of a given type in the specified resource set.

RsrcGetDesc

Returns the following resource data in the resource descriptor format provided by the caller: the file handle of the file containing the resource, flag data specifying the file handle type, the logical file address of the resource, the size in bytes of the resource, the resource type, and the resource ID.

RsrcGetSetTypeInfo

Returns in the buffer provided, all the descriptors of the specified type in the specified resource set. Descriptors are sorted numerically by resource ID.

RsrcInitSetAccess

Initializes a buffer provided by the user as an RWA for the specified run file or resource binary file.

RsrcSessionEnd

Closes an open temporary file created in a previous call to *RsrcSessionInit*. If any modified resource sets are still outstanding, *RsrcSessionEnd* calls *RsrcEndSetAccess* on them, causing the data referenced in the RWA to be written to the target files.

RsrcSessionInit

Creates a temporary file to be used by subsequent resource file operations. The caller must allocate a buffer to be used by the resource operations for file I/O.

Accessing Resources in Memory

GetModuleResourceInfo

Accesses the specified run file resource at runtime and returns information about it in the caller-provided buffer.

Comparing Strings

FComparePointer

Compares two logical addresses. *FComparePointer* returns TRUE if the addresses have the same binary value.

FsCanon

Translates a byte string into the file system canonical form with respect to case.

NlsGetYesNoStrings

Returns the strings defined in the NLS Yes or No Strings table

NlsGetYesNoStringSize

Returns the sizes of the strings defined in the NLS Yes and No Strings table.

NlsULCmpB

Compares two strings, using the lowercase to uppercase conversion table, if present, to carry out the case-insensitive comparison. *ULCmpB* returns 0FFFFh if the two strings are equal; otherwise, it returns a word containing the index of the first two characters in the strings that are different.

NlsYesNoOrBlank

Is similar to *NlsYesOrNo*, except that, in addition, *NlsYesNoOrBlank* checks for a null string.

NlsYesOrNo

Performs a case-insensitive string comparison against nationalized words meaning yes and no.

StringsEqual

Compares two strings using a boolean function that returns TRUE if the two strings are the same.

ULCmpB

Is the same as *NlsULCmpB*. (See the description of *NlsULCmpB*.)

WildCardMatch

Checks a string against a wild card specification. TRUE is returned if the string matches the specification.

Customizing the User Interface

EnlsFieldEdit

Edits a string of characters in the video display. Input to the string is derived from characters typed at the keyboard. The characters typed can be keys defining edit commands or character data to be incorporated into the string. Editing is guided by an edit structure containing information such as the location and size of the string to be edited, the cursor position, whether or not to insert characters, and the exit character finishing the editing session. New applications should use this operation rather than FieldEdit for ease in nationalization.

EnlsFieldEditByChar

Edits a string of characters in the video display. Input to the string is derived from characters provided by the caller, one character at a time. The editing performed by this operation is guided by an edit structure in the same format as the one used by EnlsFieldEdit. (See the description of EnlsFieldEdit for more information.)

FieldEdit

Displays and edits a line of text in the video display.

FormEdit

Displays a form and processes on a field-by-field basis the user-entered string response or the choice the user selected.

MenuEdit

Displays a menu from which the user can select one or more items, and processes the user response.

QueryZoomBoxPosition

Determines the location of the upper left corner of a box created with the ZoomBox operation.

QueryZoomBoxSize

Returns the size of the box created with the ZoomBox operation.

UnzoomBox

Collapses a box drawn with the ZoomBox operation.

ZoomBox

Creates a new box or increases the size of an existing box in a video display. The operation expands the box from the screen center until the specified box dimensions are reached.

Directing Data to a Byte Stream

NPrint

Prints a string to the video or other device.

OutputBytesWithWrap

Outputs a string to the video byte stream. If the string does not fit on the current line, a carriage return and tab are inserted to continue the string output on the next line.

OutputQuad

Prints a quad (32 bit, unsigned integer) to the video or other device as specified by the NPrint and PutChar operations.

OutputWord

Prints a word (16 bit, unsigned integer) to the video or other device as specified by the NPrint and PutChar operations.

PrintFileClose

Closes the file opened by the PrintFileOpen operation.

PrintFileOpen

Opens a specified file in append mode to output a disk byte stream to the file as well as to the video device.

PrintFileStatus

Returns or sets the current status of the file opened by the PrintFileOpen operation.

PutByte

Prints a byte (8 bit, unsigned integer) to the video or other device as specified by the NPrint and PutChar operations.

PutChar

Prints a character to the video or other device.

PutPointer

Prints a memory address to the video or other device as specified by the NPrint and PutChar operations.

PutQuad

Prints a quad to the video or other device as specified by the NPrint and PutChar operations.

PutWord

Prints a word to the video or other device as specified by the NPrint and PutChar operations.

SbPrint

Prints a string, in which the first byte is the size of the string. The string is printed to the video or other device as specified by the NPrint and PutChar operations.

ZPrint

Prints a null-terminated string to the video or other device as specified by the NPrint and PutChar operations.

Handling Commands

CloseSysCmds

Closes the command file and releases the buffers allocated by InitSysCmds.

GetSysCmdInfo

Returns information about a specified command name or command name abbreviation.

InitSysCmds

Opens an Executive command file of the form
{Node}[Volume]<Directory>FileName for the subsequent retrieval of
command information.

Managing Names***NameAllocClass***

Returns a unique 16-bit identifier for a class.

NameQuery

Returns the 32-bit tag value associated with the specified name in
the specified class.

NameRegister

Associates the 32-bit tag provided with the specified name and adds
this entry to the specified class in the name heap.

NameRemove

Removes a name from the specified class.

Manipulating Error Messages***CloseErcFile***

Closes an error message file on disk that was previously opened
using the *InitErcFile* operation.

GetErc

Retrieves an error message from an error message file and places
the message in memory.

GetErcLength

Returns the length of an error message.

GetFileErc

Retrieves an error message from a message file and places the error
message in memory in the format *:FileName:ErrorText*.

GetStandardErcMsg

Accepts a status code (erc) and returns the corresponding error message into a caller-supplied buffer. The message can be defined in the caller's message file.

InitErcFile

Opens the binary error message file *[Sys]<Sys>ErcMsg.bin* (distributed with Standard Software) for the subsequent retrieval of error code text by operations such as *GetErc*, *GetErcUnexpanded*, and *GetFileErc*.

PrintErc

Retrieves an error message from a message file and places the expanded message in a user-supplied byte stream.

Obtaining the System Date and Time

CompactDateTime

Converts from an expanded date/time format to system date/time format.

ExpandDateTime

Converts from the system date/time format to an expanded date/time format in which year, month, day of month, and so on, are represented as discrete fields.

FormatDateTime

Is the same as *NlsFormatDateTime*. *FormatDateTime* is documented for historic reasons only.

FormatTime

Converts an expanded date/time structure into an ASCII string containing the day, date, and time.

FormatTimeDt

Converts an expanded date/time structure into an ASCII string containing the date.

FormatTimeTm

Converts an expanded date/time structure into an ASCII string containing the time.

GetDateTime

Returns the current date/time in the system date/time format.

NlsFormatDateTime

Converts from date/time format to textual string format. This operation is used if you are creating your own NLS tables to be linked with your program. (For details on NLS, see the section entitled "Native Language Support.")

NlsParseTime

Converts a string into an expanded date/time structure.

NlsStdFormatDateTime

Obtains the memory address of the Date and Time Formats table. This operation is recommended over either NlsFormatDateTime or FormatDateTime for ease in nationalization.

ParseTime

Is the same as NlsParseTime. NlsParseTime should be used for ease in nationalization.

SetDateTime

Sets the date/time of the operating system.

SetDateTimeMode

Changes the mode of input and display of date/time strings processed by FormatTime and ParseTime.

Obtaining Versions of System Services

AccessVersion

Returns the version of the Command Access Service, including the number of log file sectors and whether logging is active.

CdVersion

Returns the CD-ROM Service version level, hardware environment (workstation or shared resource processor), and the number of CD-ROM drives currently installed.

CfaFFVersion

Returns a structure that identifies the version of the installed Cluster File Access (CFA) File Filter Service.

CfaServerVersion

Returns a structure that identifies the version of the Cluster File Access (CFA) Service installed at the server.

CfaWaVersion

Returns a structure that identifies the version of the installed Cluster File Access (CFA) Workstation Agent Service and the configuration file name used to install the service.

DcxVersion

Returns a structure that identifies the version of the installed data communications (DCX) service.

McrVersion

Returns the version number of the installed Magnetic Card Reader (MCR) service.

MouseVersion

Returns a structure that identifies the version of the installed Mouse Service.

QueueMgrVersion

Returns a structure that identifies the version of the Queue Manager, indicates if a cache is to be used, and specifies the maximum number of dynamic queues available.

RkvsVersion

Returns a structure that identifies the version of Cluster View (Remote Keyboard/Video Service) and whether the Remote User Manager is being used.

ScreenPrintVersion

Returns the version number and the size of the internal buffer for the Screen Print Service.

SeqAccessVersion

Returns version information for the specified sequential access device. This operation also returns the names of all the devices under control of the Sequential Access Service responsible for this device. (For details on the Sequential Access Service, see the *CTOS Programming Guide*.)

SpoolerVersion

Returns a structure that identifies the version of the Spooler, indicates whether to print initial form feed characters, and specifies the name of the spooler configuration file.

StatisticsVersion

Returns a structure that contains version of the installed Performance Statistics Service.

TsVersion

Returns a structure that contains the version of the installed Voice/Data Services.

XbifVersion

Returns a structure that contains the version of the installed X-Bus Interface Service.

XC002Version

Returns a structure that contains the version of the installed XC002 Service.

Parsing Configuration Files

Parsing Nonstandard Configuration Files

ConfigCloseFile

Closes a configuration file opened previously by `ConfigOpenFile`.

ConfigGetNextToken

Returns the next string in the configuration file that begins and ends with the specified characters. `ConfigFileGetNextToken` begins its search at the position last reached by a previous configuration operation.

ConfigGetPosition

Returns the position within the configuration file (previously opened by `ConfigOpenFile`) where the next read operation is to start. This position is either the beginning of the file or the point last reached by the most recent file read operation.

ConfigGetRestOfLine

Returns the contents of a configuration file (previously opened by `ConfigOpenFile`) from the file position last accessed by a previous operation to the end of the current line.

ConfigOpenFile

Opens a configuration file, allowing it to be read by subsequent configuration file operations, such as `ConfigGetNextToken`, `ConfigGetRestOfLine`, `ConfigGetPosition`, `ConfigSetPosition`, and `ConfigCloseFile`.

ConfigSetPosition

Sets the position within the configuration file (previously opened by `ConfigOpenFile`) where the next file read operation is to start. The position must be either 0 or a value recorded by a previous `ConfigGetPosition` operation.

Parsing Standard Configuration Files

LookUpField

Reads from a file and searches for a :FieldName: string entry, beginning at the current location in the file. The operation returns the string :FieldName: and the string value following the trailing colon, and the length (in bytes) of each string.

LookUpNumber

Reads from a file and searches for a :FieldName: string entry, beginning at the current location in the file. The operation returns the decimal numeric value following the string entry.

LookUpReset

Resets the point from which a scan begins to the beginning of the current file. LookUpReset should be called each time a new file is read.

LookUpString

Reads from a file and searches for a :FieldName: string entry, beginning the search at the current location in the file. The string value found after the trailing colon is returned to the caller along with the length (in bytes) of the string.

LookUpValue

Performs the same function as LookUpNumber but is more flexible. The operation can parse and return a 1, 2, or 4 byte decimal or hexadecimal numeric value.

OpenUserFile

Opens an operator's user configuration file and returns the user file handle. If the file is opened in modify mode, OpenUserFile first saves a copy of the file as a -old file.

ReadToNextField

Reads from a file and searches for a :FieldName: string. The operation saves the text strings between the current location and the :FieldName: string and sets the beginning of :FieldName: to be the current location.

SetField

Searches a file for a :FieldName: string entry and updates the string value found following the trailing colon.

SetFieldNumber

Searches a file for a string entry of the form :FieldName: and updates the string value found following the trailing colon with the specified number translated into a string. If the entry is not found, the :FieldName: string and translated number are appended to the file.

Parsing User Configuration Files

GetUserFileEntry

Opens the operator's user configuration file, searches for a :FieldName: string entry, closes the file, and returns the string value it finds just following the trailing colon of the entry.

SetUserFileEntry

Opens an operator's user configuration file, searches for a string entry of the form :FieldName:, updates the string value found following the trailing colon, and then closes the file.

Using Workstation Hardware IDs

GetClusterId

Returns the client workstation's stored ID and a list of all the modules for all the workstations in the cluster at the specified memory address.

OldReadHardId

OldReadHardId is a library version of the request, ReadHardId, and should be used for backwards compatibility with older BTOS operating systems. See ReadHardId.

OldWriteHardId

OldWriteHardId is a library version of the request, WriteHardId, and should be used for backwards compatibility with older BTOS operating systems. See WriteHardId.

ReadHardId

Retrieves a value (1 to 126) for a workstation from an add-on hardware ID device or a workstation such as the SuperGen Series 2000 or SuperGen Series 5000 that supports the request.

WriteHardId

Stores a value (1 to 126) for a workstation in an add-on hardware ID device or a workstation such as the SuperGen Series 2000 or SuperGen Series 5000 that supports the request.

Performing Miscellaneous Tasks***CopyFile***

Creates a copy of the current contents of an open file.

CreateExecScreen

Sets up the video display in the same style as the Executive screen.

GenResString

Generates an ASCII string that describes the resolution of the caller's monitor.

QueryMail

Displays a message to the video status line indicating to the user that mail is waiting. QueryMail can be called by any program.

SignOnLog

Is issued by the SignOn program to validate user access after validation of access to and contents of the user file and before user access to the system is allowed.

TextEdit

Edits a line of text. The operation takes a character and a text descriptor and returns the descriptor with appropriate changes.

WriteLog

Writes a variable-length record to the system log file.

Section 27

System Definitions

System Definitions in This Section

This section presents the system structures and other kinds of system-related information. This section also recommends methods you can use to obtain system information.

Table 27-1 presents the system structures and provides a brief description of each. (See the section entitled "System Structures," in the *CTOS Procedural Interface Reference Manual* for a detailed description of each structure.)

Table 27-1. System Structures

System Structure	Definition
80286 task state segment extension**	Is the Unisys-defined portion of the Intel structure that saves the state of a task (process or interrupt handler) in protected mode on an 80286-based microprocessor.
80386 task state segment extension**	Performs the same functions as the 80286 task state segment extension described above but is for 80386-based microprocessors.
Allowed states table	A keyboard translation or emulation supporting table used to identify chords influencing the translation or emulation of a given key.
Application system control block	Passes parameters and other information between programs within a partition.

continued

**This structure is for internal use only.

Table 27-1. System Structures (cont.)

System Structure	Definition
Batch control block*	Stores Batch Manager control information while the Batch Manager chains to a different run file in the application partition.
Boot block	Contains the information passed to the operating system by the bootstrap ROM.
Cache entry descriptor	Contains all the information unique to an entry in a cache pool.
Cache pool descriptor	Is the root structure of the cache data structures for a cache pool.
Chord information table	Is a keyboard support table in an emulation and a translation data block that defines the state of chord keys. The information includes the raw key post value, the emulated value, whether the key is defined to toggle, and whether it has an LED.
COFF section descriptor**	Describes a common object file format (COFF) file section used by the Audio Service. (For details, see the <i>CTOS Programming Guide</i> .)
Command masks table	A keyboard translation supporting table identifying the translation tables in which a specified key is defined as a command.
Communications configuration descriptor	Defines the communications byte stream dynamically by the caller of <code>AcquireByteStream</code> or through parameters in the communications configuration file when <code>OpenByteStreamC</code> is called.
Communications status buffer	Contains usage statistics for the server and the workstations attached to it.

continued

*This structure is for restricted use only.

**This structure is for internal use only.

Table 27-1. System Structures (cont.)

System Structure	Definition
Conditions table	A keyboard translation or emulation supporting table identifying the translation or emulation table that ultimately defines the key based on the state of the keyboard.
Control chords table	A keyboard translation support table identifying chord keys with different common effects.
CPU description table*	Contains shared resource processor configuration, routing, and initialization information and a pointer to an area reserved for inter-CPU communication (ICC) buffers.
Data control structure	Controls data call characteristics such as baud rate, parity, and line control for the Telephone Service.
Decoding offsets table	Is a keyboard translation supporting table containing an ordered array of offsets from the start of the translation data block to each entry in the decoding table.
Decoding table	Is a keyboard translation supporting table that specifies the unencoded keystrokes responsible for producing a particular character code.
Device control block*	Describes the type, characteristics, and status of a disk.
Diacritic masks table	A keyboard translation or emulation supporting table identifying the respective translation or emulation tables in which a specified key is defined as the first key of a diacritic pair.
Diacritics table	A keyboard translation or emulation supporting table used to identify the respective character code(s) or unencoded value(s) produced when a diacritic key pair is pressed.

continued

*This structure is for restricted use only.

Table 27-1. System Structures (cont.)

System Structure	Definition
Digital system process (DSP) control structure**	Contains commands and the address of the control program used by the DSP. (For details, see the documentation on the Audio Service in the <i>CTOS Programming Guide</i> .)
Emulation data block header	Contains the offsets to and sizes of the keyboard emulation tables and supporting tables and the identification number of the currently attached keyboard.
Emulation LEDs table	Is an emulation data block supporting table that specifies which LEDs on the actual keyboard will be turned on when specific bits in the LED word are set.
Emulation table	Keyboard table that maps the chord state and unencoded values on the source keyboard to the chord state and unencoded values necessary to produce the same character on the target keyboard.
Expanded date/time format	Contains discrete fields for the date/time, including the year, month, day of month, and so forth.
Extended partition descriptor	Contains specifications for the current application program and the exit run file.
Extended process control block	Is an extension of the process control block, containing fields for specific applications, such as the X-Bus service and MS-DOS.
File header block	Contains information about a file, its disk address, and its disk extents.
Frame descriptor	Contains all information about a video frame.
High Sierra directory record	Describes the format of a High Sierra directory record used by the CD-ROM Service. (For details, see the <i>CTOS Programming Guide</i> .)

continued

**This structure is for internal use only.

Table 27-1. System Structures (cont.)

System Structure	Definition
ISO directory record	Describes the format of an ISO directory record used by the CD-ROM service. (For details, see the <i>CTOS Programming Guide</i> .)
Log file record format	Is the format of a log file record. Each record describes an event that is significant in the cluster operation.
Low memory allocation	Contains hardware and software interrupt and trap vectors for real mode.
Operating system flags	Contains flags set at initialization to indicate the type of hardware upon which a program is running.
Partition configuration block	Contains the addresses of the extended partition descriptor, batch control block, application system control block, and the extended user structure.
Partition descriptor*	Contains the partition name, the boundaries of the partition and of its long- and short-lived memory areas, and internal links to partition descriptors in other partitions (limited internal use by real mode operating systems only).
Partition information block	Contains information associated with a given user number such as the partition configuration block selector, the descriptor table (LDT or GDT) being used, and the amount of memory allocated.
Partition keyboard information structure*	Contains keyboard data for the currently executing user number. For example, it contains the addresses of the translation and emulation data blocks, state information on diacritic sequences, string expansion information, the keyboard LED state, nationalization data, and whether the current user is using its own copies of keyboard data blocks.

continued

*This structure is for restricted use only.

Table 27-1. System Structures (cont.)

System Structure	Definition
Performance statistics structure	Is the format clients of the Performance Statistics Service use to collect statistics about the system such as task switches and swapped partitions, disk errors, files, read/write seeks, and read/write accesses.
Port structure	Contains hardware port addresses of various devices whose memory addresses differ in various configurations.
Process control block	Contains the software context of a process.
Queue entry header	Is the first 40 bytes of each queue entry in a queue file. The header is reserved for the Queue Manager.
Queue file header	Contains all the information that the Queue Manager needs to manage the entries in the queue.
Queue status block	Contains a queue entry's server user number, priority, and the buffers in which the queue entry handles for the queue entry and the logically following queue entry are stored.
Resource descriptor	Contains the size, lfa, type code, and ID of a resource.
Repeat attributes table	A keyboard translation supporting table defining typematic keys and keys for which the User Bit is set.
Special keys table	A keyboard translation and emulation supporting table that contains the raw unencoded values of all the special keys such as CANCEL , FINISH , ACTION , LEFT SHIFT , and RIGHT SHIFT .
Spooler queue entries	Are the formats of a spooler control queue, status queue, and scheduling queue entries. For details on the Spooler Service, see the <i>CTOS Programming Guide</i> .
Standard file header format	Contains file header information, such as the file signature, file type, and the minimum and maximum file record sizes.

continued

Table 27-1. System Structures (cont.)

System Structure	Definition
Standard record header format	Contains record information, such as the unique record identifier, the physical record size, and the record status.
Standard record trailer format	Indicates whether the record is malformed.
String masks table	Is a keyboard emulation or translation supporting table identifying the respective translation or emulation tables in which a specified key will generate a string.
String offsets table	Is a keyboard emulation or translation supporting table containing offsets from the start of the emulation or translation data block to each string entry in the Strings supporting table.
Strings table	Is a keyboard emulation or translation supporting table containing string entries. Each entry is indexed by the key post value of the key generating the string.
System configuration block	Contains detailed information about the system image.
System date/time structure	Contains information about the system date and time to a greater precision than 1 second.
Telephone Service configuration block	Describes the configuration of the Telephone Service.
Telephone Service configuration file format	Is the format of the Telephone Service configuration information.
Telephone status structure	Describes the state of the Voice Processor hardware and all users. The information is obtained by calling the TsGetStatus operation.
Terminal output buffer	Used by the shared resource processor (SRP) terminal management operations.
Timer pseudointerrupt block	Used by the SetTimerInt and ResetTimerInt operations. (See "Timer Management," for details on these operations.)

continued

Table 27-1. System Structures (cont.)

System Structure	Definition
Timer request block format	Controls the realtime clock (RTC) services. (See "Timer Management," for details on these operations.)
Translation data block header	Contains the memory addresses and sizes of the keyboard translation tables and supporting tables and the identification number of the currently attached keyboard.
Translation table	Maps a raw or emulated key post value to the displayable character code.
User control block	Contains the default node, default volume, default directory, and default password set by the last SetPath operation, and the default file prefix set by the last SetPrefix operation.
Variable length parameter block	Communicates parameters when a program chains to another program.
Video control block	Contains all information known to the operating system about the video display.
Voice control structure	Provides the Telephone Service with the information it needs to perform a recording or playback.
Voice file header	Is the first 512 bytes of a file used by the voice interface of Voice/Data services. Using a CODEC, voice messages can be recorded to the file and later played back.
Voice file record	Is the format of the voice recording sectors for the voice interface of Voice/Data services.
Voice Processor control block	Contains information used to coordinate the operations of the Telephone Service and the Voice Processor module.
Volume home block*	Is the root structure (that is, the starting point for the tree structure) of the information that describes a file system on a disk volume.

continued

*This structure is for restricted use only.

Methods of Obtaining System Information

Certain operations provide access to particular system structures. These operations and the system structures your program can access are as follows:

Operation	System Structure
GetPAscb	Application system control block (ASCB)
GetPartitionStatus	Partition information block
GetUcb	User control block (UCB)
GetVhb	Volume home block (VHB)

Use the GetPStructure operation to access the system structures not listed individually above.

Programs that access a system structure directly are not compatible with operating systems executing in protected mode.

As an example, historically, the video control block (VCB) could be accessed directly by its memory address (244h) in low memory. This required a segment address of 0. The resulting logical address thus was

0:244

In protected mode, this address implies a selector (SN) of 0. An SN with a value of 0, however, is invalid. (For guidelines on writing protected mode programs, see the *CTOS Programming Guide*.)

GetPStructure provides your program with a valid memory address compatible in real mode and in protected mode.

System Definition Operations

The system information operations described below are categorized by use. Operations are arranged alphabetically in each group. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Cluster Management

GetClusterStatus

Returns usage statistics for each communications line and for the workstations attached to it.

Disk Management

QueryDiskGeometry

Returns disk geometry information for the specified device to the memory area provided.

SetDiskGeometry

Sets the disk geometry for the specified device.

File Management

GetVhb

Copies the VHB of the specified device to the specified memory area.

Operating System

CurrentOsVersion

Determines the version of an operating system. *CurrentOsVersion* should be used instead of *OsVersion* for programs that run on earlier versions of the operating system.

EnterBootRom

Transfers control to the beginning of the boot ROM.

FilterDebugInterrupts

Directs the Debugger to pass through Debugger interrupts (single step, breakpoint) on a per process basis by sending messages to an exchange.

FMergedOs

Returns TRUE if the caller is running on a CTOS II 3.3 or later operating system. TRUE also means the GetModuleId operation provides X-Bus+ support.

FProcessorSupportsProtectedMode

Returns TRUE on protected mode microprocessors such as the 80286 or 80386. It returns FALSE on an 8086 or 80186 microprocessor.

FProtectedMode

Returns TRUE if the calling program is executing in protected mode. It returns FALSE if the program is executing in real mode.

FRmos

Returns TRUE if the caller is executing in real mode and the operating system is executing in protected mode.

FRmosUser

Is the same as FRmos, except FRmosUser allows the specification of a user number.

FVSeries

Returns TRUE if the caller is executing on a CTOS III operating system.

GetCoproprocessorStatus

Reports if either a math coprocessor or a software floating-point emulator, such as the Math Service, is present to execute floating-point instructions.

GetFRmosUser

Is used to determine a client's execution mode. GetFRmosUser sets the flag *FRmos* to TRUE if the specified user number is executing a real mode program. Otherwise, the flag is FALSE.

GetLocalDaiNumber

Returns the device address identification (DAI) number for the workstation on which the caller is running.

GetPartitionStatus

Returns status information about the specified application partition and the program currently executing in it.

GetPAscb

Returns the address of the ASCB of the application partition in which the program is executing.

GetPStructure

Returns the memory address of an operating system structure.

OsVersion

Is the same as CurrentOsVersion. CurrentOsVersion, however, should be used for programs running on earlier versions of the operating system.

QueryCoproprocessor

Reports if a coprocessor, such as the Math Service, is present to execute floating-point instructions.

QueryLdtR

Returns the global descriptor table (GDT) selector that identifies the specified user number's local descriptor table (LDT). If the user number does not have an LDT, QueryLdtR returns the null selector.

QueryNodeName

Obtains the node name of the local node where this request is issued.

SerialNumberOldOsQuery

Returns the boot ROM's unique 32-bit serial number at the specified address.

SerialNumberQuery

Returns the boot ROM's unique 32 bit serial number at the specified address. For backwards compatibility, use `SerialNumberOldOsQuery` instead of this operation.

SetPStructure

Provides controlled write access to selected fields of certain system data structures that may legitimately be modified by user programs running in protected mode.

User Name Management

GetUcb

Copies the UCB for the current user to the specified area.

GetUserStatus

Copies user status information to the specified memory area.

GetWsUserName

Returns the user name that is signed on at the specified cluster workstation.

QueryUserLocation

Returns information about the specified user number in the memory area provided.

SetWsUserName

Stores the user SignOn name of the workstation.

Video

QueryVideo

Performs the same function as `QueryVidHdw`, except `QueryVideo` fills in all fields in the specified memory area.

QueryVidHdw

Places information describing the level of video capability of a workstation in the specified memory area.

Index

- !Sys, 12-28
- \$ Directory, 12-42
- +Sys, 12-29
- .def file (*See* module definition file)
- 16-bit addressing, 24-1
- 32-bit addressing, 24-1
- 80286 task state segment extension, 27-1
- 80386 task state segment extension, 27-1
- :fSuppressGlobalPolicy:, 3-14

A

- abort requests, 33-18
- accessing
 - and modifying system date and time, 26-22
 - commands, 26-18
 - communication ports, 15-2
 - disk devices, 13-1
 - files, 12-4, 12-8, 12-28, 20-2, 20-4
 - global data, 39-12
 - memory of another processor, 42-5
 - NLS tables, 45-17
 - resources in disk files, xli, 26-2 to 26-14
 - resources in memory, xlii, 26-14
 - system services, 4-7, 30-30, 30-31 to 30-32
 - volume home blocks, 13-4
 - X-Bus module memory, 41-3
 - X-Bus modules in protected mode, 41-5
 - X-Bus modules in real mode, 41-5
- accessing system services
 - using the kernel primitives, 4-7, 30-31 to 30-32
 - using the request procedural interface, 30-30
- accessing NLS tables
 - alternate tables linked with application, 45-17
 - tables in application memory, 45-17
 - tables loaded at system initialization, 45-17
- AcquireByteStreamC, 15-5, 15-6
- Action, 10-34, 11-12, 11-45, 11-52 to 11-53
- adding passwords to files, 12-13
- adding resources to files, 26-11
- address mapping, 39-2
- addressability, 36-11
- addressing
 - 16-bit, 24-4
 - 32-bit, 24-4
 - a byte in a segment, 24-3
 - memory, 4-10
 - protected mode, 24-3
 - real mode, 24-3
 - segment base addresses, 24-3
- addressing memory, 4-10
- addressing, 16-bit, 24-4
- addressing, 32-bit, 24-4

- advantages of dynamic link
 - libraries, 39-21
- advantages of using
 - device-dependent interfaces, 8-2
 - sequential access method, 8-1
- aliasing, 34-5
- AllocAllMemory, 24-13
- AllocAllMemorySL, 24-9
- AllocAreaSL, 24-9, 24-13
- AllocateSegment, 2-13, 24-11, 24-17
- allocating
 - response exchanges, 30-31
 - global linear address space, xl
 - memory, 2-22
- AllocCommDmaBuffer, 42-6, 42-8
- AllocExch, 30-54, 33-9
- AllocHugeMemory, 2-13, 24-11, 24-12, 24-17
- AllocMemoryFramesSL, 24-13
- AllocMemoryLL, 24-9, 24-14, 6-8
- AllocMemorySL, 24-9, 24-13, 33-9
- allowed states, 11-2, 11-23
- allowed states table, 27-1
- alternate request procedural
 - interface, 30-30
- application profile keyboard, 11-2, 11-43
- application system control block, 6-3, 27-1, 36-5
- ASCB (See application system control block)
- AsGetVolume, 35-14
- Assembler, 4-1
- AsSetVolume, 35-14
- assigning
 - I/O addresses to X-Bus modules, 41-1
 - process priorities, 2-2
 - volume passwords, 12-12
- AssignKbd, 36-22
- AssignVidOwner, 36-22
- asynchronous operation, 20-1
- asynchronous RS-232-C
 - communications, 15-4
- Asynchronous System Service
 - operations
 - AllocMemoryInit, 35-2
 - AsyncRequest, 35-2
 - AsyncRequestDirect, 35-2
 - BuildAsyncRequest, 35-2
 - BuildAsyncRequestDirect, 35-2
 - CheckContextStack, 35-2
 - CreateContext, 35-2
 - HeapAlloc, 35-2
 - HeapFree, 35-2
 - HeapInit, 35-3
 - LogMsgIn, 35-3
 - LogRequest, 35-3
 - LogRespond, 35-3
 - ResumeContext, 35-3
 - SwapContextUser, 35-3
 - TerminateAllOtherContexts, 35-3
 - TerminateContext, 35-3
 - TerminateContextUser, 35-3
- AtFileInit, 12-44
- AtFileNext, 12-44
- atomic, 31-3, 31-5
- Audio Service operations
 - AsSetVolume, 35-14

automatic data segment, 39-13
automatic target mode functions,
 18-21
automatic volume recognition,
 12-11
automatic pause between full text
 frames, 10-5, 10-7
AVR (*See* automatic volume
 recognition)

B

backing store, 3-2
Batch control block, 27-2, 36-5
Beep, 11-56
binary mode, 8-7
binary resource file, 26-3
binding (*See* linking)
binding system-common procedures,
 4-4
bit-map workstations, 10-1
blocked records, 20-1
blocks
 storing copies of request blocks,
 32-2
 transferring ICC messages, 32-5
boot block, 27-2
bootable volumes, 43-1
bsKbd, 8-3, 8-4
bsVid, 8-3, 8-4
BSWA (*See* byte stream work area)
buffer ownership table, 32-6, 32-7
buffers, 25-3
 DAM, 23-2
 caller-supplied byte stream, 8-3
 ownership table, 32-6, 32-7
 management modes, 23-3

BuildFileSpec, 12-37 to 12-38,
 12-48
BuildFullSpecFromPartial, 12-48
building
 and parsing file specifications,
 12-35 to 12-38
 customized operating systems,
 43-1
 request blocks, 33-4
 single-line text editors, 26-27
 system keyboard files, 11-32
building and parsing file
 specifications, 12-35 to 12-38
BuildSpecFromDir, 12-48
BuildSpecFromFile, 12-48
BuildSpecFromNode, 12-48
BuildSpecFromPassword, 12-48
BuildSpecFromVol, 12-48
built-in networking, 1-2, 1-4
bus address, 42-1
bus addresses, 32-2, 32-6
 on shared resource processors,
 42-3 to 42-5
 users, 42-5
byte stream work area, 8-1
byte streams, 8-3, 8-4, 9-1

C

cache entry descriptor, 25-5, 27-2
cache pool descriptor, 25-4, 27-2
cache pool handle, 25-4
cache statistics block, 25-13
CacheClose, 25-13, 25-14
CacheFlush, 25-12, 25-14
CacheGetEntry, 25-9, 25-14
CacheGetStatistics, 25-13, 25-14
CacheGetStatus, 25-12, 25-14

Index

- CacheInit, 25-14
- CacheReleaseEntry, 25-9, 25-14
- caching, 1-4, 1-5
- calculating cache entries, 25-6
- calculating request sizes, 32-9
- call gates, 34-7
- calling DLL procedures, 39-11
- CdAbsoluteRead, 35-5
- CdAudioCtl, 35-5
- CdClose, 35-5
- CdControl, 35-4
- CdDirectoryList, 35-4
- CdGetDirEntry, 35-4
- CdGetVolumeInfo, 35-4
- CdOpen, 35-5
- CdRead, 35-5
- CdSearchClose, 35-4
- CdSearchFirst, 35-4
- CdSearchNext, 35-4
- CdServiceControl, 35-5
- CDT (See CPU description table)
- CdVerifyPath, 35-4
- CdVersion, 26-38
- CD-ROM Service operations
 - CdAbsoluteRead, 35-5
 - CdAudioCtl, 35-5
 - CdClose, 35-5
 - CdControl, 35-4
 - CdDirectoryList, 35-4
 - CdGetDirEntry, 35-4
 - CdGetVolumeInfo, 35-4
 - CdOpen, 35-5
 - CdRead, 35-5
 - CdSearchClose, 35-4
 - CdSearchFirst, 35-4
 - CdSearchNext, 35-4
 - CdServiceControl, 35-5
 - CdVerifyPath, 35-4
- CfaFFVersion, 26-38
- CfaServerVersion, 26-38
- CfaWaVersion, 26-38
- Chain, 5-8, 33-11
- change user number requests,
 - 33-20
- ChangeCommLineBaudRate, 16-3,
 - 16-5
- ChangeFileLength, 12-7, 12-22,
 - 12-44
- ChangeOpenMode, 12-46
- ChangePriority, 33-9
- changing
 - file passwords, 12-13
 - and querying SCSI path
 - parameters, 18-9
 - keyboards, 11-42
 - or removing a directory password,
 - 12-12
 - processor modes, 1-6
- character
 - attributes, 10-2
 - cell sizes, 10-14
 - code, 11-2
 - extended set, 11-3
 - mode, 8-8, 11-2
 - multibyte, 11-4
 - plus, 11-12, 11-13
 - repeating, 45-6
 - standard, 11-4
- character attributes, 10-2
- character code, 11-2
- character mode, 8-8, 11-2
- character plus, 11-12, 11-13
- character repeating, 45-6
- character set
 - extended, 11-3
 - standard, 11-4

- character-map workstations, 10-1
- Check, 30-17, 30-54
- CheckErc, 5-6, 5-7
- CheckForOperatorRestartC, 15-5, 15-7
- checking SCSI operation error
 - status, 18-16 to 18-18
- CheckpointBs, 8-16
- CheckpointBsAsyncC, 15-8
- CheckPointBsC, 15-7
- CheckpointRsFile, 22-3
- CheckpointSysIn, 11-57
- CheckReadAsync, 12-49, 30-30
- checksum, 29-4
- CheckWriteAsync, 12-49, 30-30
- chord information structure, 27-2
- chord state, 11-2
- chords, 11-2, 11-23
- chords supporting table, 11-30 to 11-31
- cleaning pages, 3-2, 3-11
- cleaning up resources, 39-15
- clearing semaphores, 31-3, 31-8
- ClearPath, 12-45
- clients, 30-1, 30-2, 30-53, 33-1, 34-4, 39-1
- clock algorithm, 3-9 to 3-11
- CloseAllFiles, 12-22, 12-43
- CloseAllFilesLL, 12-22, 12-46
- CloseAltMsgFile, 45-34
- CloseByteStream, 8-15
- CloseDaFile, 23-4
- CloseErcFile, 26-20, 26-21, 26-35
- CloseFile, 12-22, 12-43
- CloseMsgFile, 45-33
- CloseRsFile, 22-3
- CloseRtClock, 37-3, 37-8
- CloseServerMsgFile, 45-26, 45-35
- CloseSysCmds, 26-18, 26-34
- CloseVidFilter, 8-4, 9-5
- closing
 - cache pools, 25-13
 - command files, 26-18
 - error message files, 26-21
 - files, 12-26
- closing cache pools, 25-13
- closing command files, 26-18
- closing error message files, 26-21
- closing files, 12-26
- CLRB (*See* communications line return block)
- cluster, 2-6
 - communications channels, 44-1
 - configuration, 2-6, 30-32, 30-33, 44-1
 - processor, 2-7
 - workstations, 2-6, 2-7
- cluster communications channels, 44-1
- cluster processor, 2-7
- CMIH (*See* communications mediated interrupt handlers)
- code segment, 24-4, 24-6
- code sharing, 1-7, 29-1
- COFF section descriptor, 27-2
- collapsing boxes, 26-16
- color programming, 10-17
- Comm Nub, 40-16
- Command Access Service operations
 - ObtainAccessInfo, 35-6
 - ObtainUserAccessInfo, 35-6
- command
 - bit, 11-16
 - case, 6-5
 - form, 6-2
 - interpreter (*See* Executive)
 - masks table, 27-2

- command interpreter (*See* Executive)
- command masks table, 27-2
- commands
 - obtaining information about, 26-18
 - opening a command file, 26-18
- common object file format, 27-2
- communication between application partitions, 36-17
- communications
 - byte streams, 8-8
 - channels, 2-6, 5-6, 15-3, 16-2
 - channel identifiers, 8-11
 - configuration descriptor, 27-2
 - line return block, 16-2
 - parameters, 15-5
 - port expander specifications, 8-12
 - programming, 15-1 to 15-5
 - raw interrupt handlers, 40-16
 - status buffer, 27-2
- communications line return block, 16-2
- communications programming, 15-1 to 15-5
- communications mediated interrupt handlers, 40-18, 40-19
- communications raw interrupt handlers, 40-16
- communications status buffer, 27-2
- CompactDateTime, 26-22, 26-36
- comparing
 - DLLs to other CTOS services, 39-3
 - logical addresses, 26-28
 - strings, 26-14 to 26-15
- compatibility, 45-6
- concurrent file access, 2-5
- conditions table, 27-3
- conditions, 11-23
- ConfigCloseFile, 26-25, 26-40
- ConfigGetNextToken, 26-25 to 26-26, 26-40
- ConfigGetRestOfLine, 26-25, 26-40
- ConfigOpenFile, 26-25, 26-40
- ConfigQueryFilePosition, 26-25, 26-26
- ConfigQueryPosition, 26-40
- ConfigSetFilePosition, 26-25, 26-26
- ConfigSetPosition, 26-40
- configuration files, 8-10 to 8-10
- configuring
 - line speeds, 44-1
 - system paging parameters, 3-14
 - systems to use DLLs, 39-16
- context switch, 29-3, 29-4, 34-5
- contiguous memory regions,
 - managing 24-11
- control chords table, 27-3
- control register, 16-2
- ControlInterrupt, 40-30
- controlling
 - character attributes, 10-5, 10-6
 - character repeat rate, 11-45
 - external interrupt occurrences, 40-8
 - frame allocation, 3-14
 - screen attributes, 10-5
 - scrolling and cursor positioning, 10-5, 10-6
 - semaphore wait semantics, 31-13
- controlling character attributes, 10-5, 10-6
- converting between date and time formats, 26-22
- converting NLS keyboard tables, 11-32
- ConvertToSys, 33-10, 33-28, 34-8, 36-9

- copy data file resources, 26-11, 26-13
- CopyFile, 26-28, 26-43
- copying files, 26-28
- copying multiple resources, 26-12
- CP (*See* cluster processor)
- CP boards, 17-1
- CParams, 6-4, 6-10
- CPU description table, 27-3, 32-2, 32-6
- Crash, 5-7
- Create Keyboard command, 11-16
- Create Keyboard Data Block
 - command, 11-26
- CreateAlias, 24-15
- CreateBigPartition, 36-8, 36-12, 36-15, 36-22
- CreateDir, 12-7, 12-45
- CreateExecScreen, 26-43
- CreateFile, 12-7, 12-9, 12-23, 12-43
- CreatePartition, 36-8, 36-12, 36-22
- CreateProcess, 33-9
- CreateUser, 36-22
- creating
 - bit-map fonts, 10-1
 - bootable volumes, 43-1
 - character-map fonts, 10-1
 - loadable request files, 33-16 to 33-17
 - module definition files, 39-10, 39-14
 - SCSI paths, 18-6
- creating a SCSI path
 - changing and querying path parameters, 18-9
 - default timeout, 18-8
 - defining unique paths, 18-6, 18-7
 - disconnection permission, 18-8
 - read-only SCSI path parameters, 18-9
 - specifying path parameters, 18-8
 - using the path handle, 18-9
- creating and accessing files
 - using byte streams, 12-20, 12-24, 12-26
 - using file management operations, 12-20, 12-22, 12-24, 12-25 to 12-26
 - using structured file access
 - methods, 12-20
- creating bit-map fonts, 10-1
- creating character-map fonts, 10-1
- creating loadable request files, 33-16 to 33-17
- creating module definition files, 39-10, 39-14
- creating partitions, 36-8, 36-15 to 36-16
- CRIH (*See* communications raw interrupt handlers)
- critical section semaphores, 31-3, 31-5, 31-8
 - as an alternative to disabling interrupts, 31-8
 - compared to noncritical, 31-9
 - effect on suspending processes, 31-10
- critical section, 31-5
- CSubParams, 6-4, 6-10
- CTOS foundation
 - event-driven, priority-ordered process scheduling, 1-2
 - messaged-based operation, 1-2
 - multiprogramming, 1-2
 - multitasking, 1-2
 - nationalization, 1-2
 - networking, 1-2

- CTOS I, xliii, 1-1
- CTOS II, xliii, 1-1
- CTOS III, xliii, 1-1, 1-7
- CTOS III enhancements
 - demand paging, 1-7
 - dynamic link libraries, 1-7
 - name management, 1-8
 - semaphores, 1-8
- CTOS, 1-2
- Ctos.lib, xxxviii
- CTOS/XE, 1-2 to 1-2
- CtosToolkit.lib, xxxviii
- CurrentOsVersion, 4-15, 27-10
- customizing
 - keyboard data blocks, 10-37, 11-31
 - to 11-36
 - SAM object modules, 8-3
 - system services, 2-4
 - user interfaces, 26-16 to 26-16
- customizing data blocks
 - application-specific blocks, 11-32
 - by converting NLS keyboard tables, 11-32

D

- DAM (*See* direct access method)
- DAM buffer, 23-2
- data control structure, 27-3
- data files, 26-3, 26-9
- data processor, 2-7
- data register, 16-2
- data segment selector
 - expand down, 24-5
 - expand up, 24-5
- data, sharing, 29-1
- DAWA (*See* direct access work area)
- DCB (*See* device control block)
- DCXVersion, 26-38
- DDS (*See* Digital data storage)
- DeallocAliasForServer, 24-16
- DeallocateRods, 38-17
- DeallocateSegment, 24-11, 24-17
- deallocating
 - DMA buffers, 42-7
 - system resources, 5-6
 - user numbers, 36-8
- DeallocExch, 30-55
- DeallocHugeMemory, 24-11, 24-12, 24-17
- DeallocMemoryLL, 24-9, 24-14
- DeallocMemorySL, 24-9, 24-13
- DeallocRunFile, 36-18, 36-23
- DeallocSg, 24-15
- decoded value, 11-2
- decoding offsets table, 27-3
- decoding table, 27-3
- default devices, 8-2
- default response exchange, 30-2, 30-18
- DefineInterLevelStack, 24-14
- DefineLocalPageMap, 24-14
- defining
 - diacritic key pairs, 11-40
 - DLL segments, 39-17
 - SRP request routing, 32-3 to 32-5
 - system service requests, 30-27
 - toggle keys, 11-40
 - unique SCSI paths, 18-6, 18-7
- Delay, 37-2, 37-8
- delaying address mapping, 39-9
- DeleteByteStream, 9-5
- DeleteDaRecord, 23-4
- DeleteDir, 12-45
- DeleteFile, 12-43

- demand paging, xl, 1-7, 3-4, 2-20, 42-3
- determining
 - error message length, 26-20
 - monitor resolution, 26-28
 - existence of I-Bus devices, 41-2
- development utilities
 - Assembler, 4-1
 - Librarian, 4-1, 34-11
 - Linker, 2-13, 4-1
- device
 - access to disk files, 13-1
 - control blocks, 12-35, 27-3
 - handler processes, 40-7
 - handlers, 2-5, 40-5
 - interrupt handlers, 40-7
 - interrupts, 40-3
 - names, 13-2
 - passwords, 12-8, 12-13, 13-2
 - specifications, 13-2
- device control block, 12-35, 27-3
- device/file specifications, 8-10 to 8-13
- DeviceInService, 40-30
- devices
 - default, 8-2
 - masking, 40-10
 - patience, 40-9
- device-dependent
 - access to the video, 10-3
 - interfaces, 8-2
 - programs, 7-3
- device-independent
 - access to the video, 10-3
 - programs, 7-3
- device-specific operations, 9-2
 - SetImageMode, 9-2
 - GetBsLfa, 9-2
 - PutBackByte, 9-2
 - QueryVidBs, 9-2
 - SetBsLfa, 9-2
- diacritic keys, 11-15
- diacritic masks table, 27-3
- diacritic, 11-3, 11-15
- diacritical key handling, 45-7
- diacritics table, 27-3
- digital data storage, 8-9
- digital system process, 27-4
- direct access method, 12-3, 20-4, 23-1
- direct access work area, 23-2
- directing data to a byte stream, 26-16 to 26-17
- directory
 - creating, 12-7
 - defined, 12-7
 - maximum files, 12-7
 - names, 12-7
 - passwords, 12-8, 12-12
 - protection, 12-7
 - specifications, 12-9
- directory passwords, 12-8, 12-12
- directory specifications, 12-9
- dirty pages, 3-2
- DisableActionFinish, 11-57
- DisableCluster, 44-4
- disabling interrupts, 31-8
- DiscardInputBsC, 15-6
- DiscardOutputBsC, 15-6
- disk
 - byte streams, 8-5
 - devices, 13-2
 - partitions, 13-4
- DismountVolume, 13-5
- dispatching
 - communications interrupt handlers, 40-16
 - dynamic link library cleanup procedures, 39-15

- distributed environment, 2-6
- DLL (See dynamic link library)
- DMA buffer mapping, 42-3
- DmaMapBuffer, 42-6, 42-7, 42-9
- DmaUnmapBuffer, 42-9
- doorbell interrupt, 32-2, 32-7, 32-9, 32-11, 32-12
- Doze, 37-2, 37-8
- DP (See data processor)
- DSP control structure, 27-4
- duplication of volume control structures, 2-5
- dynamic binding (See dynamic linking)
- dynamic data segment, 24-5
- dynamic link libraries, xlii, 1-7, 39-1
 - advantages, 39-21
 - and the Linker, 39-5
 - cleaning procedures, 39-15
 - compared to request-based services, 39-6
 - compared to system-common services, 39-4 to 39-6
 - execution model, 39-4
 - guidelines for writing, 39-12 to 39-15
 - initializing procedures, 39-14
 - using resources, 39-11
- dynamic linking, 2-24
 - at runtime, 39-20 to 39-21
 - how it works, 39-6 to 39-8
 - impact on loading, 39-8 to 39-9
 - loadtime, 39-1
 - runtime, 39-1
- dynamically customizing keyboard data blocks, 11-33

- dynamically installable services, 2-3
- system-common procedures, 34-1
- dynamically redirecting video byte streams, 10-5, 10-6

E

- EAR, 41-5 (See extended address register)
- economizing on
 - memory usage, 39-4
 - processor time, 31-5
- editing a submit file, 11-50
- emulating LEDs, 11-31
- emulating, 11-3
- emulation
 - defined, 11-28
 - examples, 11-28 to 11-29
 - data block headers, 27-4
 - data blocks in writable segments, 2-23
 - LEDs table, 27-4
- emulation data blocks, 11-26
- emulation LEDs table, 27-4
- emulation table, 27-4
- EnableSwapperOptions, 38-17
- enabling and disabling interrupts, 40-9
- enclosures, SRP, 32-2
- encrypted passwords, 12-18
- encrypting volumes, 12-18
- ending a resource session, 26-8
- end-of-interrupt, 40-10
- ENLS (See extended native language support)
- ENLS operations, 45-19 to 45-22

- Enls.lib, xxxviii
- EnlsAppendChar, 45-31
- EnlsCase, 45-20, 45-30
- EnlsCbToCCols, 45-22, 45-32
- EnlsClass, 45-20, 45-30
- EnlsDeleteChar, 45-31
- EnlsDrawBox, 45-32
- EnlsDrawFormChars, 45-32
- EnlsDrawLine, 45-32
- EnlsFieldEdit, 26-32, 45-21, 45-31
- EnlsFieldEditByChar, 26-16, 26-32, 45-32
- EnlsFindC, 45-21, 45-32
- EnlsFindRC, 45-21, 45-32
- EnlsGetChar, 45-22, 45-31
- EnlsGetCharWidth, 45-31
- EnlsGetPrevChar, 45-31
- EnlsInsertChar, 45-32
- EnlsMapCharToStdValue, 45-20, 45-31
- EnlsMapStdValueToChar, 45-20, 45-31
- EnlsQueryBoxSize, 45-32
- EnterBootRom, 27-10
- EOI (*See* end-of-interrupt)
- ercOK, 4-3
- error
 - codes (*See* status codes)
 - handling, 5-6
 - message file, 26-20
 - reporting, 4-3
- error codes (*See* status codes)
- ErrorExit, 5-7, 33-11
- ErrorExitString, 5-7
- escape sequences, 11-49
- establishing
 - directory passwords, 12-12
 - communications channels, 40-15
 - interrupt handlers, 40-4
 - multiplexed interrupt handlers, 40-23
- event-driven, priority-ordered
 - process scheduling, 1-2, 1-3, 2-2
- events, 2-2
- examples of
 - CTOS calls, 4-3
 - naming conventions, 4-3
 - procedural interface, 4-2
- exceptions, 40-4 (*See also* internal interrupts)
- exchange routing, 30-49 to 30-50
- exchanges 2-2, 5-6
 - allocating, 30-18
 - defined, 30-11, 30-17
 - queues, 30-22
 - response, 30-16
 - sending messages to, 30-19
 - system service, 30-16
 - types, 30-18
 - waiting for messages at, 30-21
- exclusive access to resources, 31-5
- exclusive mode (mx), 18-10
- executable program, 5-1
- executing real mode applications, 3-13
- execution, thread of, 29-1
- Executive, 2-4, 5-5, 6-1, 10-3, 10-4
- exercising administrative control over the cluster, 43-1, 44-3
- exit run files, 5-5 to 5-5
- Exit, 5-8, 33-11, 34-9
- ExitAndRemove, 5-8
- ExitListQuery, 39-22
- ExitListSet, 39-22
- expand down segment, 24-1

Index

- expand up segment, 24-1
- ExpandAreaLL, 24-9, 24-14
- ExpandAreaSL, 24-9, 24-13
- ExpandDateTime, 26-22, 26-36
- expanded date/time format, 26-22, 27-4
- expanding boxes, 26-16
- expanding specifications, 30-42 to 30-43
- ExpandLocalMsg, 45-33
- explicitly enabling SCSI target mode functions, 18-21
- exports, 39-2
- extended address register, 41-5
- extended character sets, 11-3, 45-5
- extended native language support, 1-5
- extended partition descriptor, 27-4, 36-5
- extended process control block, 27-4
- extended system services, xxxviii, 28-3, 35-1 to 35-3
- extending memory, 4-14
- external interrupt handler
 - examples, 40-22
- external interrupt handling model, 40-5
- external interrupts, 40-2, 40-3
- extracting parameter data from ASCB, 6-3

F

- FAB (See file area blocks)
- far procedures, 38-7, 39-14
- FatalError, 5-7
- faulting pages into physical memory, 38-2

- faults, 40-26
- FCB (See file control blocks)
- FComparePointer, 26-28, 26-31
- features, operating system, xxxvii
- FHB (See file header blocks)
- FieldEdit, 26-32
- file access modes, 2-5
- file area blocks, 12-24, 12-34
- file control blocks, 12-24, 12-34
- file handles, 12-22, 12-24
- file header blocks, 12-6, 27-4, 12-23
- file management, 2-5
 - accessing files, 12-1
 - augmenting, 12-3
 - automatic volume recognition, 12-3
 - capabilities, 12-1
 - concurrent file access, 12-1
 - creating a file, 12-22
 - device independence, 12-21
 - hierarchical organization, 12-1
 - performing I/O to a disk file, 12-25
 - throughput capability of disk hardware, 12-1
 - using Nls.sys, 12-1
 - volume encryption, 12-3
- file password, 12-8, 12-13
 - adding passwords to files, 12-13
 - changing file passwords, 12-13
- file processor, 2-7
- file protection
 - assigning file passwords, 12-13
 - assigning volume passwords, 12-12
 - by password, 12-11
 - changing or removing directory passwords, 12-12
 - directory passwords, 12-12

- establishing directory passwords, 12-12
- volume encryption, 12-11
- volume passwords, 12-12
- file protection by protection level, 12-14 to 12-17
- file specifications
 - abbreviated, 12-10
 - default, 12-10
 - directory, 12-9
 - full file, 12-9
- files
 - access, 12-7
 - changing file length, 12-7
 - creating, 12-7
 - defined, 12-7
 - names, 12-8
 - protecting, 12-7
 - renaming, 12-7
- files, accessing using hybrid access methods, 20-5
- files, creating and accessing, 20-2
- FillBufferAsyncC, 15-4, 15-6
- FillBufferC, 15-6
- FillFrame, 10-18
- FillFrameRectangle, 10-18
- FilterDebugInterrupts, 27-11
- filtering
 - keyboard input, 11-15
 - ReadKbdInfo requests, 11-46
- filters, 2-4, 30-15, 33-8
 - defined, 33-22, 30-51
 - one-way pass-through, 33-22 to 33-23
 - replacement, 33-22
 - system requests for, 33-25
 - two-way pass-through, 33-24
 - uses, 33-25
- fixed length records, 20-2
- fixed partitions, 2-13
- fixed-length records, 20-1
- floating-point coprocessors, 40-3
- FlushBufferAsyncC, 15-6
- FlushBufferC, 15-6
- flushing cache entries, 25-12
- FMergedOs, 27-11
- Format Disk command, 13-4
- Format, 13-5
- FormatDateTime, 26-36
- FormatTime, 26-36
- FormatTimeDt, 26-36
- FormatTimeTm, 26-37
- formatting
 - cache memory, 25-6 to 25-8
 - characteristics, 19-1
 - disks, 13-4
- FormEdit, 26-32
- forms-oriented interfaces, 6-1
- ForwardRequest, 30-15, 30-55
- FP (*See* file processor)
- FProcessorSupportsProtectedMode, 27-11
- FProtectedMode, 27-11
- frame descriptor, 10-17, 27-4
- FrameBackSpace, 10-18
- frames, 3-1, 3-2
- FRmos, 27-11
- FRmosUser, 27-11
- FsCanon, 26-31
- FSrpUp, 44-4
- functions, 4-3
- FValidPbCb, 24-16
- FVSeries, 27-11

G

GDT (*See* global descriptor table)
general processor, 2-7
generating a system, 43-1
generating SAM (*See* SAMGen)
Generic Print Access Method, 14-2,
19-1
Generic Print System byte streams,
8-6
Generic Print System, 14-1
GenResString, 26-28, 26-43
GetAltMsg, 45-26, 45-34
GetAltMsgUnexpandedLength,
45-34
GetBoardInfo, 32-18
GetBsLfa, 9-2, 9-5
GetCanonicalNodeAndVol, 12-48
GetClstrGenerationNumber, 44-4
GetClusterId, 26-42
GetClusterStatus, 27-10
GetClusterStatus, 44-1, 44-4
GetCommLineDmaStatus, 16-6
GetCoprocessorStatus, 27-11
GetCParasOvlyZone, 38-16
GetCtosDiskPartition, 13-4, 13-5
GetDateTime, 26-22, 26-37
GetDirInfo, 12-46
GetDirStatus, 12-45
GetErc, 26-20, 26-21
GetErc, 26-35
GetErcLength, 26-20, 26-35
GetFhLongevity, 12-46
GetFileErc, 26-20, 26-21, 26-35
GetFileInfoByName, 12-44
GetFileStatus, 12-44
GetFRmosUser, 27-11

GetKbdId, 11-56
GetKeyboardId, 11-56
GetLocalDaiNumber, 27-12
GetModuleAddress, 41-2, 41-9
GetModuleId, 41-9
GetMsg, 45-33
GetMsgUnexpanded, 45-33
GetMsgUnexpandedLength, 45-33
GetNlsDateName, 45-28
GetNlsKeycapText, 45-28
GetNlsTable, 45-17, 45-28
GetOvlyStats, 38-16
GetPartitionExchange, 36-8
GetPartitionHandle, 36-8, 36-17,
36-20
GetPartitionStatus, 27-12, 36-8,
36-16, 36-17, 36-20
GetPartitionSwapMode, 36-21
GetPAscb, 27-12
GetPNlsTable, 45-28
GetProcInfo, 32-3, 32-18
GetPStructure, 11-33, 26-21, 27-9,
27-12, 36-17
GetRsLfa, 22-3
GetScsiInfo, 18-26
GetSegmentLength, 24-16
GetServerMsg, 45-26, 45-35
GetSlotFromName, 32-18
GetSlotInfo, 32-3, 32-18
GetStamFileHeader, 20-8
GetStandardErcMsg, 26-36
GetSysCmdInfo, 26-18, 26-34
GetUcb, 12-45
GetUserFileEntry, 26-23, 26-23,
26-42
GetUserNumber, 36-8, 36-20
GetVhb, 12-49, 27-10

- global
 - descriptor table, 4-12, 5-6, 24-3
 - initialization, 39-2, 39-14, 39-20
 - linear address space, 24-3, 24-11, 33-9
 - linear addresses, 2-16, 3-2, 3-6
 - page map, 3-5
 - segments, 39-2, 39-9
 - thrashing, 3-2, 3-9
 - global descriptor table, 4-12, 5-6, 24-3
 - global segments, 39-2, 39-9
 - GP (*See* general processor)
 - GPAM (*See* Generic Print Access Method)
 - GPS (*See* Generic Print System)
 - grouping segments, 5-3
 - guidelines for writing
 - CMIHs, 40-18 to 40-19
 - CRIHs, 40-16 to 40-17
 - MIHs, 40-22
 - RIHs, 40-20 to 40-21
- ## H
- half- and full-duplex
 - communications, 15-3
 - handling commands, 26-18
 - handling error conditions, 36-1
 - handling nationalized strings, 26-15
 - hardware dependencies below byte
 - stream interface level, 16-2
 - hash, 25-1, 25-4
 - heap, 24-1
 - hierarchical organization of files
 - node, 12-4
 - volume, 12-5
 - file specifications, 12-36
 - High Sierra directory record, 27-4
 - high-level interfaces, 4-8 to 4-9, 7-1, 7-3 to 7-4
 - high-resolution timing, 37-1
 - host adapter, 18-5
 - huge segments, 24-1, 24-4
 - defined, 4-10
 - managing, 24-11
 - hypersegment swapping, 3-4
 - hypersegments, 2-12, 36-3 to 36-5, 36-8 (*See also* partition components)
 - code, 36-7
 - in-memory relationships, 36-3 to 36-5
 - local descriptor table, 36-5
 - long-lived memory, 36-7
 - short-lived memory, 36-7
 - user structure, 36-5 to 36-6
- ## I
- I/O
 - blocks, 12-34
 - interfaces, 4-8
 - nonoverlapped, 20-2, 20-4
 - overlapped, 22-1, 22-2
 - ICC (*See* inter-CPU communication)
 - ICC request/response ring queues, 32-2
 - ICC segments, 32-6
 - ICMS (*See* Intercontext Message Service)
 - identifying unique SCSI devices, 18-7
 - identifying workstations of a given hardware type, 26-27

- IDT (*See* interrupt descriptor table)
- image mode, 8-7
- import library, 39-2
- import, 39-2
- improving program performance, 38-14
- including delimiters in strings, 26-26
- index file, ISAM, 20-2
- Indexed Sequential Access Method, 12-3, 20-2 to 20-3, 21-1
- informing user of waiting mail, 26-28
- InitAltMsgFile, 45-26, 45-34
- InitCharMap, 10-21, 10-9
- InitCommLine, 16-2, 16-5, 40-15, 40-30
- InitErcFile, 26-20, 26-36
- initializing
 - cache, 25-5 to 25-8
 - character maps, 10-9
 - dynamic link libraries, 39-10
 - resource access, 26-8
 - timer request blocks, 37-3
- initiating video refresh, 10-9
- initiator, 18-20
- InitLargeOverlays, 38-16
- InitMsgFile, 45-33
- InitOverlays, 38-16
- InitSysCmds, 26-18, 26-35
- InitVidFrame, 10-9, 10-21
- input event types, storage of, 11-4
- installing trap handlers, 40-27
- InstallNet, 30-55
- InstallSystemCommon, 34-4
- instance initialization, 39-14, 39-2, 39-20
- instance segments, 39-2, 39-9
- INT, 40-25
- interboard routing (*See* inter-CPU communication)
- Intercontext Message Service, 30-5, 36-17
- Inter-CPU communication, 2-4, 32-1
- interface levels, defined, 4-8
- intermodule, general-purpose expansion bus (*See* X-Bus)
- internal interrupt handlers, 40-28
- internal interrupts, 40-2, 40-25 (*See also* traps)
- internationalization, 45-1
 - defined, 45-2
 - extended native language support, 1-5
 - keyboards, 1-4
- internationalizing software, 43-1
 - extending internationalization, 45-19
 - guidelines, 45-18
- interprocess communication, 2-2, 30-1, 30-52 to 30-53, 33-2, 34-1
- interrupt descriptor table, 40-2
- interrupt flag, 40-8
- interrupt handlers, 40-2
 - and general processor boards, 40-13
 - and virtual memory operating systems, 40-12
- communications, 40-15 to 40-19
- defined, 40-2
- guidelines for writing CMIHs, 40-18 to 40-19
- guidelines for writing CRIHs, 40-16 to 40-17
- guidelines for writing MIHs, 40-22

- guidelines for writing RIHs, 40-20
 - to 40-21
 - nesting, 40-13
 - packaging, 40-28 to 40-29
 - styles, 40-12, 40-14
 - interrupt handling styles, 40-1
 - interrupt hierarchy, 40-3
 - interrupt latency, 31-8
 - interrupt levels, 40-2, 40-3, 40-4
 - interrupt vector table, 40-2
 - interrupts
 - defined, 40-2
 - device, 40-3
 - external, 40-2, 40-3
 - hierarchy, 40-3, 40-4
 - internal, 40-2, 40-3 (*See also* traps)
 - interrupt handlers, 40-2
 - levels, 40-2
 - lost, 40-11
 - nonmaskable, 40-12
 - pending, 40-11
 - intersegment references, 5-4
 - IPC (*See* interprocess communication)
 - IPC applications
 - communicating between
 - application partitions, 30-5
 - communicating within an
 - application partition, 30-4
 - managing resources, 30-7
 - synchronizing processes, 30-6
 - IPC components, 30-11
 - IPC extension (*See* inter-CPU communication)
 - IPC messages (*See* request blocks)
 - IRET, 40-26
 - ISAM (*See* Indexed Sequential Access Method)
 - ISAM data set, 20-2 to 20-3, 21-1
 - ISAM.lib, 20-3
 - ISO directory record, 27-5
 - IVT (*See* interrupt vector table)
 - I-Bus
 - device management, 11-54
 - disconnecting drivers, 11-55
 - installing loadable drivers, 11-55
 - protocol, 11-55
 - reading input events, 11-55
 - I-Bus device management, 11-54 to 11-55
 - I-Bus style keyboard, 11-3
 - I-key, 11-3, 11-20
- ## K
- kernel functions, 2-1
 - kernel primitives, 4-4
 - kernel, 2-1
 - key post value, 11-9, 11-11
 - keyboard
 - application profile, 11-2, 11-43
 - configuration options, 11-43
 - I-Bus style, 11-3, 11-13
 - mapping IDs, 11-44
 - PC-style, 11-4, 11-13
 - setting options, 11-45
 - source, 11-31
 - supporting multiple profiles, 11-43
 - system profile, 11-4, 11-41
 - target, 11-31
 - emulation LEDs table, 11-31
 - keyboard and video independence, 11-54
 - keyboard byte streams, 8-8

- keyboard code, 11-3
- keyboard customization, 45-5
- keyboard data block, 11-3, 11-18 to 11-26
 - customizing, 11-31 to 11-36
 - general layout, 11-18
 - translation, 11-20
 - organization, 11-18 to 11-23
- keyboard event, 11-3
- keyboard hardware protocols, 11-13
- keyboard interrupt service routine, 11-9
- keyboard interrupt, 11-9
- keyboard management
 - features, 45-5 to 45-6
 - international keyboard support, 1-4
 - multiple keyboard support, 1-4
 - options, 11-1
 - overview, 45-7 to 11-10
- keyboard modes, 11-10 to 11-13
 - character, 11-2, 11-12
 - character plus, 11-12
 - comparing, 10-11
 - mapped unencoded, 11-12
 - raw unencoded, 11-9, 11-12
 - unencoded, 11-4, 11-11
 - unencoded plus, 11-12
- keyboard postprocessing, 11-10
- keyboard preprocessing, 11-9
- keyboard state, 11-10
- keyboard subtables, 10-16
- keyboard supporting tables, 11-18, 11-19, 11-22, 11-23
 - allowed states table, 11-23
 - allowed states, 11-23
 - chords table, 11-23
 - chords, 11-23
 - command masks table, 11-25

- conditions table, 11-23
- conditions, 11-23
- decoding table, 11-26
- defining toggle chords, 11-40
- diacritic key handling, 11-40
- diacritic masks table, 11-25
- multibyte masks table, 11-25
- multibyte strings, 11-40
- keyboard translation table, 11-22
- KeyboardProfile, 11-57
- keyboards
 - customizating, 1-5
 - source, 1-5
 - target, 1-5

L

- LaFromP, 24-16
- LaFromSn, 24-16
- language definition, 45-2
- large model, 24-6
- late binding (*See* dynamic linking)
- LDT (*See* local descriptor table)
- least-recently-used, 25-1
- level, 12-36
- lfa (*See* logical file address)
- LfsToMaster, 12-29
- LibFree, 39-22
- LibGetHandle, 39-22
- LibGetInfo, 39-22
- LibGetProcInfo, 39-22
- LibLoad, 39-22
- Librarian, 4-1
- line speed configuration, 44-1
- linear address space, 3-7, 36-9
- linear memory addresses, 4-13
- lines, 44-1 (*See also* cluster communication channels)
- linked-in service, 2-3

- Linker, 2-13, 4-1, 5-3
 - linking
 - dynamic, 4-5
 - object modules into run files, 5-1 to 5-3, 38-1
 - programs, 5-1 to 5-3
 - static, 4-5
 - linking, dynamic, 4-5
 - linking, static, 4-5
 - loadable request files, 33-7, 34-5
 - LoadBackgroundPalette, 10-22
 - LoadColorStyleRam, 10-22
 - LoadFontRam, 10-9, 10-21
 - loading additional run files into
 - partitions, 36-16
 - loading programs, 5-4 to 5-4, 36-16
 - LoadInteractiveTask, 36-12, 36-16, 36-23
 - LoadPrimaryTask, 36-16, 36-22
 - LoadRunFile, 36-18, 36-23
 - LoadTask, 36-18, 36-23
 - local descriptor table, 4-12, 36-3, 36-5, 38-8
 - local linear addresses, 2-17, 3-2, 3-6
 - local page map, 3-5
 - local resource-sharing networks
 - (See cluster)
 - local semaphores (See RAM semaphores)
 - local thrashing, 3-2, 3-9
 - localization, 45-1, 45-23
 - locating the CTOS disk partition, 13-4
 - locating the volume home block, 13-4
 - locked pages, 3-2
 - LockIn, 16-5
 - LockInContext, 36-21
 - locking a noncritical semaphore, 31-7
 - locking pages, 3-12
 - locking semaphores, 31-3
 - LockOut, 16-5
 - LockVideo, 10-23
 - LockVideoForModify, 10-23
 - LockXbis, 41-9
 - log file record format, 27-5
 - logical file address, 12-21
 - logical memory addresses
 - defined, 4-12
 - example, 4-12
 - logical unit number, 18-5, 18-7
 - long-lived memory, 2-22 to 2-23, 6-3, 24-10, 36-3, 36-7
 - long-lived, 24-2
 - LookUpField, 26-24, 26-41
 - LookUpNumber, 26-24, 26-41
 - LookUpReset, 26-24, 26-41
 - LookUpString, 26-24, 26-41
 - low memory allocation, 27-5
 - low resolution timing, 37-1
 - low-level interfaces, 4-8 to 4-9, 7-1
 - LRU (See least-recently used)
 - LUN (See logical unit number)
 - macros
 - using to distinguish between keys in the standard character set, 45-15
 - using to expand messages, 45-26
- ## M
- MakePermanent, 38-16
 - MakePermanentP, 38-17
 - MakeRecentlyUsed, 38-17

- managing
 - data packets, 18-22
 - names, 26-18 to 26-20
 - physical memory, 2-13, 2-19
 - resources by system services, 28-2
 - system date and time, 26-21
- manipulating error messages, 26-20 to 26-21
- MapBusAddress, 42-1, 42-3, 42-5, 42-8
- MapCsIOvly, 38-17
- MapIOvlyCs, 38-17
- mapped unencoded, 11-9, 11-12
- mapping
 - addresses between clients and system services, 34-5
 - device-independent operations to device-dependent, 9-1
 - keyboard IDs, 11-44
 - linktime to runtime addresses, 39-9
- MapPStubPProc, 38-17
- MapXBusWindow, 41-4, 41-9
- MapXBusWindowLarge, 41-5, 41-9
- maskable, 40-8
- masking device priority levels, 40-10
- master agent, 30-33
- master file directory, 12-6
- matching keyboards to data blocks, 11-42
- McVersion, 26-38
- mediated interrupt handlers, 40-13, 40-22
- MediateIntHandler, 40-30
- mediating buffer wait conditions, 32-6, 32-14 to 32-17
- medium model, 24-6, 34-6
- memory
 - allocating for code and static data, 24-2
 - long-lived, 24-10, 24-2
 - partition, 24-2
 - short-lived, 24-2
- memory addresses
 - logical memory addresses, 4-12
 - physical, 4-10
 - translating, 4-10 to 4-11
- memory disk, 13-3, 25-3
- memory management styles, xliii, 2-11 to 2-13
- memory master, 41-3
- memory organization
 - application partition, 2-22 to 2-23
 - system memory, 2-14 to 2-18
 - application partition, 24-8 to 24-9
- memory slave, 41-3
- memory use, maximizing, 25-4
- memory, short-lived, 5-6
- MenuEdit, 26-32
- merging requests, 33-7, 33-16
- message file facility
 - alternate message file operations, 45-26
 - creating and editing message files, 45-24
 - defined, 45-24
 - standard message file operations, 45-26
 - system service message file
 - operations, 45-26
 - using a single message file, 45-26
 - using macros with messages, 45-26
 - using multiple message files, 45-26
 - using very few messages, 45-26

- message files, 43-1
- message work area, 26-20, 45-24
- messaged-based operation, 1-2, 1-3, 2-2
- messages, 30-23 (*See also* request blocks)
 - kernel primitives for receiving, 30-16 to 30-17
 - kernel primitives for sending, 30-12 to 30-15
- MIH (*See* mediated interrupt handlers)
- mode 3 DMA, 41-6
- Mode3DmaReload, 41-9
- models of computation, 5-3, 24-6, 34-6, 39-13
- module definition file, 39-2, 39-10
- MountVolume, 13-5
- Mouse Services operations
 - GetIBusDevInfo, 35-7
 - PDAssignMouse, 35-8
 - PDGetCursorPos, 35-7
 - PDGetCursorPosNSC, 35-7
 - PDInitialize, 35-6
 - PDLoadCursor, 35-7
 - PDLoadSystemCursor, 35-7
 - PDQueryControls, 35-7
 - PDQuerySystemControls, 35-7
 - PDReadCurrentCursor, 35-7
 - PDReadIconFile, 35-7
 - PDSetCharMapVirtual
 - Coordinates, 35-6
 - PDSetControls, 35-8
 - PDSetCursorDisplay, 35-8
 - PDSetCursorPos, 35-8
 - PDSetCursorPosNSC, 35-8
 - PDSetCursorType, 35-6
 - PDSetMotionRectangle, 35-6
 - PDSetMotionRectangleNSC, 35-6
 - PDSetSystemControls, 35-8
 - PDSetTracking, 35-6
 - PDSetVirtualCoordinates, 35-6
 - PDTranslateNSCtoVC, 35-8
 - PDTranslateVCToNSC, 35-8
 - ReadInputEvent, 35-7
 - ReadInputEventNSC, 35-7
- Mouse Services, 10-3, 11-14
- MouseVersion, 26-38
- MoveFrameRectangle, 10-18
- MoveOverlays, 38-16
- moving program segments between
 - disk and memory, 38-1
- multibyte characters, 11-4, 45-1, 45-5
- multibyte strings, 11-4, 11-16
- multipartition memory
 - management, 2-12, 2-14 to 2-16, 3-3, 3-4, 4-13
- multipartition operating systems, 36-1, 36-9, 36-11, 38-1
- multiprogramming, 1-2, 1-3, 28-1, 28-2, 36-1, 36-6
- multitasking, 1-2, 1-4
- multithreading (*See* multitasking)
- multi-instance system services, 33-21
- mutual exclusion on CTOS, 31-10
- mutual exclusion semaphores, 31-3, 31-5
- mutual exclusion semaphores using
 - Send and Wait, 31-11 to 31-12
- MWA (*See* message work area)

N

- name management, xli, 1-8
- named addresses, 39-6
- naming conventions, xlv to xlviii, 4-2 to 4-3
- nationalizable operations, 26-16
- nationalization, 1-2, 1-4, 45-5
- nationalizing programs, 4-15
- nationalizing the system date and time, 26-22
- native language support, 45-1 to 45-35
- native language support tables
 - accessing the functionality of, 45-6
 - character class, 45-12
 - Clustershare key post values, 45-15
 - ClusterShare keyboard extended codes, 45-15
 - Clustershare keyboard, 45-14
 - Clustershare video translation, 45-15
 - collating sequence, 45-10
 - Context Manager, 45-16
 - date and time formats, 45-9
 - date name translations, 45-10
 - file system case, 45-8
 - keyboard chords, 45-13
 - keyboard mapping, 45-7
 - keycap legends, 45-9
 - lowercase to uppercase, 45-8
 - multibyte escape keys, 45-14
 - NLS Strings, 45-14
 - number and currency formats, 45-10
 - source file, 45-2
 - special characters, 45-13
 - uppercase to lowercase, 45-8
 - video byte streams text, 45-8
 - yes or no strings, 45-13
- near procedure, 38-7
- nesting interrupt handlers, 40-10
- Net agent, 30-34, 30-51
- Net server, 30-34
- network configuration, 30-32
- network environment, 2-11
- network routing, 30-34, 30-45, 30-46 to 30-47
- network, 2-6
- networking, built-in, 1-2, 1-4
- NLS keyboard tables, 11-33
- NlsCase, 45-28
- NlsClass, 45-29
- NlsCollate, 45-29
- NlsFormatDateTime, 26-22, 26-37, 45-29
- NlsGetYesNoStrings, 26-15, 26-31, 45-29
- NlsGetYesNoStringSize, 26-15, 26-31, 45-29
- NlsKbd.sys, 1-5, 11-18, 11-19, 11-34 to 11-35, 45-5 (*See also* system keyboard file)
- NlsNumberAndCurrency, 45-29
- NlsParseTime, 26-22, 26-37, 45-29
- NlsSpecialCharacters, 45-29
- NlsStdFormatDateTime, 26-22, 26-37, 45-29
- NlsULCmpB, 26-15, 26-31, 45-30
- NlsVerifySignatures, 45-30
- NlsYesNoOrBlank, 26-15, 26-31, 45-5, 45-30
- NlsYesNoStrings, 45-5
- NlsYesNoStringSize, 45-5
- NlsYesOrNo, 26-15, 26-31, 45-30, 45-5

node names, reserved, 30-42
node, defined, 30-32
noncritical semaphores, 31-3, 31-6,
31-7
nonterminatable semaphores, 31-3,
31-10
normal mode, 8-7, 11-46
NPrint, 26-17, 26-33
null process, 29-3, 29-4

O

object module procedures, 4-4
ObtainAccessInfo, 35-6
obtaining
cache entries, 25-9
cache statistics, 25-13
cache status, 25-12
cluster status, 44-1
command information, 26-18
partition status, 36-17
physical memory addresses, 42-7
system information, 27-9
system data and time, 26-21 to
26-22
ObtainUserAccessInfo, 35-6
offset, 4-12, 24-3 to 24-4 (*See also*
relative address)
OpenByteStream, 8-15
OpenByteStreamC, 15-5, 15-6
OpenDaFile, 23-4
OpenFile, 12-8, 12-24, 12-43
OpenFileLL, 12-22, 12-24, 12-46
opening
command files, 26-18
files, 12-24
system semaphores, 31-6 to 31-6
byte streams, 8-3
OpenNlsFile, 45-30
OpenRsFile, 22-3
OpenRtClock, 37-3, 37-8
OpenServerMsgFile, 45-26, 45-35
OpenUserFile, 26-24, 26-41
OpenVidFilter, 8-4, 9-5
operating system features, xxxvii
operating system flags, 27-5
operating system types, 2-7 to 2-10
operational modes, 1-1
operations
kernel primitives, 4-4, 4-6
object module procedures, 4-4
object module procedures, 4-5
request-based, 4-7
system-common procedures, 4-5
using the request procedural
interface to system services,
4-4
optimizing
a cache, 25-10
performance, 3-11, 25-13
organizing parameter data in
ASCB, 6-3
OsVersion, 4-15, 27-12
OutputBytesWithWrap, 26-17,
26-33
OutputQuad, 26-17, 26-33
OutputToVid0, 8-15
OutputWord, 26-17, 26-33
outstanding requests, 30-32
overlay descriptors, 38-6
overlay zone header, 38-6
overlays, 38-2 (*See also* virtual code
management)
overrun, 40-11
oversubscribing memory, 2-21, 3-2,
3-3, 3-8

P

- pages, 3-1, 3-3
 - cleaning, 3-3
 - faults, 40-26
 - mapping, 3-5 to 3-5
 - replacement, 3-9 to 3-11
- paging parameters, 3-14
- paging service, 2-13, 2-16
- paging service, components, 3-12
- paragraph, 24-3, 24-9
- parallel port interrupt handlers, 40-22
- parameters, 6-2
- ParseFileSpec, 12-37 to 12-38, 12-47
- ParseSpecForDir, 12-47
- ParseSpecForFile, 12-47
- ParseSpecForNode, 12-47
- ParseSpecForPassword, 12-47
- ParseSpecForVol, 12-47
- ParseTime, 26-37
- parsing
 - answers to Executive yes/no options, 26-15
 - configuration files, 26-23 to 26-26
 - device/file specifications, 8-14
 - nonstandard configuration files, 26-25
 - specifications, 16-2
 - standard configuration files, 26-24 to 26-25
- partition
 - components, 33-11
 - defined, 5-1
 - configuration block, 27-5, 36-5
 - descriptor, 27-5
 - handle (See user number)
 - information block, 27-5
 - keyboard information structure, 27-5
 - partition configuration block, 27-5, 36-5
 - partition descriptor, 27-5
 - partition handle (See user number)
 - partition information block, 27-5
 - partition keyboard information structure, 27-5
 - partition managing programs, 28-3, 36-1, 36-12
 - partition memory, allocating and deallocating, 24-9 to 24-10 (See also short-lived and long-lived memory)
 - partition, 28-3
 - variable length parameter block structure, 6-3, 6-4
 - swapping, 3-4, 36-13 to 36-14
 - partitioned memory, 36-1
- partitions, 2-11
 - application, 36-2
 - components, 36-3 to 36-7
 - disk, 13-4
 - fixed, 36-2
 - system, 36-2
 - types, 36-2
 - variable, 36-2, 36-7
 - with multiple run files, 36-18
- passwords
 - device, 12-8
 - directory, 12-8
 - file, 12-8
 - specifying, 12-8
 - types, 12-8
 - volume, 12-8

- password protection
 - encrypted, 2-5
 - protection level, 2-5
- password protection to SCSI
 - devices, 18-5
- path handle, 18-9
- patience, 40-9, 40-27
- PC emulation, 1-1
- PDGetCursorPos, 35-7
- peek mode (mp), 18-10
- performance, optimizing I/O to
 - sequential access devices, 8-10
- Performance Statistics structure, 27-6
- Performance Statistics System
 - Service operations
 - ClosePsSession, 35-9
 - DeinstPsServer, 35-9
 - GetPsCounters, 35-9
 - OpenPsLogSession, 35-9
 - OpenPsStatSession, 35-9
 - PsCloseSession, 35-9
 - PsDeinstServer, 35-9
 - PsGetCounters, 35-9
 - PsOpenLogSession, 35-9
 - PsOpenStatSession, 35-9
 - PsReadLog, 35-10
 - PsResetCounters, 35-10
 - ReadPsLog, 35-10
 - ResetPsCounters, 35-10
- performance, 40-12
- periodically checking for messages, 30-32
- physical address space, 4-10
- physical devices, directly
 - controlling, 7-3
- physical memory address, defined, 4-13, 4-10
- physical records, 20-6
- piecemealing DMA buffers, 42-7
- PIT (See programmable interval timer)
- pixels, 10-1
- placing
 - clients in wait state, 30-2
 - error messages in user-supplied byte streams, 26-21
 - information in video control blocks, 10-9
- playback mode, 11-47
- polling, 44-2
- port structure, 27-6
- porting applications to CTOS, 31-2
- porting programs to different native languages, 4-15
- PosFrameCursor, 10-18
- posting data blocks, 11-17, 11-36, 11-42, 11-54
- PostKbdTable, 11-36, 11-57
- predefined byte stream work areas, 8-4
- prefaulting pages, 3-3, 3-11
- preopened byte streams, 8-3
- pre-GPS spooler byte streams, 8-7
- primary partition, 33-11, 33-20
- primary task, 36-18, 38-15
- PrintAltMsg, 45-34
- printer byte streams, 8-5
- PrintErc, 26-20, 26-21, 26-36
- PrintFileClose, 26-17, 26-33
- PrintFileOpen, 26-17, 26-33
- PrintFileStatus, 26-17, 26-33
- printing modes, 8-7
- PrintMsg, 45-33
- prioritizing interrupt signals, 40-9
- procedural interface, 4-2

- process, 2-1, 28-2
 - context switch, 29-3
 - context, 29-2
 - defined, 29-1
 - preempting execution, 29-4
 - priorities, 29-2, 29-3, 30-53
 - scheduling, 29-2
 - states, 29-4
- process control block, 27-6, 29-3
- process priorities, 29-2, 29-3, 30-53
- process, preempting execution of, 29-4
- processes
 - client, 2-3
 - system service, 2-3
- processor modes, changing 1-6
- ProcInfoNonres, 38-7
- ProcInfoRes, 38-6
- producer/consumer model
 - using CTOS kernel primitives, 31-15 to 31-16
 - using semaphores, 31-13 to 31-15
- program, 36-18
 - exceptions, 40-25
 - performance, 11-16, 25-3, 38-14
 - portability, 26-16
 - termination, 5-5 to 5-5, 11-54
- ProgramColorMapper, 10-22
- programmable interrupt controller, 40-9
- programmable interval timer, 28-3, 37-1, 37-6
- programmatic interface, 4-2
- programming documentation, 4-1
- programming tools (*See also* development utilities)
 - Linker, 24-6, 24-9
 - Resource Compiler, 26-3
 - Resource Librarian, 26-3
- programs and run files, 2-10 to 2-11
- prompts, 6-2
- protected mode addressing, 4-10, 4-13
- protected mode advantages, 4-14
- protected mode enhancements
 - caching, 1-5
 - extended native language support, 1-5
 - multiple and international keyboard support, 1-5
 - protected mode operation, 1-6
 - real mode application support, 1-6
- SCSI management, 1-6
- variable partitions with code sharing, 1-7
- protected mode operation, 1-4
- protected mode programs, 5-4
- protection by protection level, 12-14 to 12-17
- protection levels, 12-14 to 12-17
- PSend, 30-55, 40-5, 40-30
- pseudointerrupt, 40-24
- public procedures, 38-6
- PurgeMcr, 11-59
- PutBackByte, 9-2, 9-5
- PutByte, 26-17, 26-34
- PutChar, 26-17, 26-34
- PutCharsAndAttrs, 10-19
- PutFrameChars, 10-19
- PutFrameCharsAndAttrs, 10-19
- PutPointer, 26-17, 26-34
- PutQuad, 26-17, 26-34
- PutWord, 26-17, 26-34

Q

QIC (*See* quarter-inch cartridge)

quarter-inch cartridge, 8-9

QueryBigMemAvail, 24-9, 24-15

QueryBoardInfo, 32-18

QueryBounds, 10-19

QueryCharsAndAttrs, 10-19

QueryCoproprocessor, 27-12

QueryCursor, 10-19

QueryDaLastRecord, 23-4

QueryDaRecordStatus, 23-4

QueryDcb, 27-11

QueryDefaultRespExch, 30-54

QueryDeviceName, 13-5

QueryDeviceNames, 13-5

QueryDiskGeometry, 13-5

QueryExitRunFile, 5-8

QueryFrameAttrs, 10-20

QueryFrameBounds, 10-19

QueryFrameChar, 10-20

QueryFrameCharsAndAttrs, 10-20

QueryFrameCursor, 10-20

QueryFrameString, 10-20

querying

communications parameters,
15-5

paging statistics, 3-13

parameter data in ASCB, 6-4

QueryIOOwner, 24-15

QueryKbdLeds, 11-56

QueryKbdState, 11-58

QueryLdtR, 27-12

QueryMail, 26-28, 26-43

QueryMemAvail, 24-9, 24-15

QueryModulePosition, 41-9

QueryNodeName, 33-28

QueryPagingStatistics, 3-13, 3-15

QueryRequestInfo, 33-8, 33-10,
33-28

QueryTrapHandler, 40-28, 40-30

QueryVidBs, 9-2, 9-5, 10-8

QueryVideo, 10-21

QueryVidHdw, 10-9, 10-20

QueryWsNum, 44-4

QueryZoomBoxPosition, 26-32

QueryZoomBoxSize, 26-33

queue entry header, 27-6

queue file header, 27-6

Queue Manager operations

AddQueue, 35-10

AddQueueEntry, 35-10

CleanQueue, 35-10

DeinstallQueueManager, 35-10

EstablishQueueServer, 35-10

GetQmStatus, 35-10

MarkKeyedQueueEntry, 35-11

MarkNextQueueEntry, 35-11

ReadKeyedQueueEntry, 35-11

ReadNextQueueEntry, 35-11

RemoveKeyedQueueEntry, 35-11

RemoveMarkedQueueEntry,
35-11

RemoveQueue, 35-11

RescheduleMarkedQueueEntry,
35-11

RewriteMarkedQueueEntry,
35-11

TerminateQueueServer, 35-12

UnmarkQueueEntry, 35-12

queue status block, 27-6

QueueMgrVersion, 26-38

queuing messages at exchanges,
30-22

R

- RA (*See* offset; *See also* relative address)
- RAM semaphores, precautions, 31-18 to 31-19
- RAM semaphores, 31-3, 31-4
- random I/O, 20-2, 20-4
- raw interrupt handlers, 40-12
- raw unencoded mode, 11-4
- raw unencoded value, 11-9
- raw unencoded, 11-12, 11-13, 11-16
- RDT (*See* resource descriptor table)
- Read, 12-43
- ReadActionCode, 11-52, 11-58
- ReadActionKbd, 11-52, 11-58
- ReadAsync, 12-49, 30-30
- ReadBsRecord, 8-15
- ReadByte, 8-15
- ReadBytes, 8-16
- ReadByteStreamParameterC, 15-7
- ReadCommLineStatus, 16-4, 16-5, 40-30
- ReadDaFragment, 23-2, 23-5
- ReadDaRecord, 23-4
- ReadDirSector, 12-45
- ReadHardId, 26-27, 26-43
- reading and writing a file, 12-25 to 12-26
- reading data blocks
 - comparing reading methods, 11-39
 - in an object module, 2-34
 - into application memory, 11-36 to 11-38
 - reading binary file into local memory, 11-38
 - using byte streams, 11-37
 - using ReadOsKbdTable, 11-37
 - reading keyboards, 11-14
 - reading submit files, 11-14, 11-48
- ReadInputEvent, 11-14
- ReadKbd, 11-56
- ReadKbdDataDirect, 11-56
- ReadKbdDirect, 11-49, 11-57
- ReadKbdInfo, 11-12, 11-14 to 11-16, 11-55, 11-57, 45-6
- ReadKeySwitch, 32-18
- ReadMcr, 11-59
- ReadOsKbdTable, 11-37, 11-58
- ReadRsRecord, 22-3
- ReadStatusC, 15-7, 16-4
- ReadToNextField, 26-24, 26-25, 26-41
- ready state, 29-4
- read-only SCSI path parameters, 18-9
- real mode addressing, 4-13
- real mode application support, 1-4, 1-6
- real mode applications, 4-10
- ReallocHugeMemory, 24-11, 24-12, 24-17
- realtime clock, 28-3, 37-1
- ReceiveCommLineDma, 16-6
- receiving
 - messages, 30-16 to 30-17
 - requests, 32-11
 - responses, 32-12
- record fragment, 23-2
- record sequential access method, 12-3, 20-3, 22-1
- record sequential work area, 22-2
- recording file, 11-49, 11-54
- recording mode, 11-49

- records
 - blocked, 20-1, 23-1
 - fixed length, 20-1, 20-2, 23-1
 - fragment, 23-2
 - number, 23-1
 - physical, 20-6
 - random access to, 23-1
 - sequential access to, 22-1
 - spanned, 20-1, 23-1
 - standard file header, 20-6
 - standard record trailer, 20-6
 - variable length, 20-1, 22-1
- redefining keys, 11-30
- redirecting video byte streams, 5-4, 10-5
- redo keystroke buffer, 6-5
- reducing disk access, 26-4
- reentrant code, 34-6, 34-8, 39-5
- referencing memory
 - physical memory, 4-14
 - static memory allocated to other programs, 4-14
- register, BR0, 40-13
- ReinitLargeOverlays, 38-16
- ReinitOverlays, 38-16
- ReinitStubs, 38-17
- related documentation, xliii to xlv
- relative address, 4-12, 24-3 to 24-4
 - (*See also* offset)
- release documentation, 43-1
- ReleaseByteStream, 8-16
- ReleaseByteStreamC, 15-8
- ReleasePermanence, 38-17
- ReleaseRsFile, 22-3
- releasing cache entries, 25-11 to 25-12
- RemakeAliasForServer, 24-16
- RemakeFh, 12-46
- RemapBusAddress, 42-8
- RemoteBoot, 32-18
- RemoveFfsBrackets, 12-48
- RemovePartition, 36-8, 36-16, 36-17, 36-23
- removing interrupt handlers, 40-19
- removing partitions, 36-17
- RenameByteStream, 9-5
- RenameFile, 12-7, 12-43
- ReopenFile, 12-46
- repeat attributes table, 27-6
- repetitive timing, 37-5
- replacing local trap handlers, 40-28
- replacing pages, 3-8, 3-9
- representing the system date and time in compact form, 26-21
- Request, events that occur when called, 30-45
- request block routing code, 30-43
- request blocks, 30-1, 30-2, 30-11, 30-12, 33-2
 - components, 30-24
 - constructing, 30-30
 - control information, 30-26
 - example of Write request, 30-28 to 30-29
 - format, 30-24
 - header, 30-25 to 30-26
 - request data item, 30-27
 - response data item, 30-27
 - routing code, 30-27
- request codes, 33-2
 - assigning to system service requests, 30-9
 - defining system requests, 30-10
 - levels, 30-9
 - linked into operating system, 30-10

- registering, 30-9
- reserved, 30-9
 - using to route requests, 30-9
- request definition, 33-7
- request procedural interface, 2-3,
30-1 to 30-4, 30-53, 33-4
 - defined, 4-6
 - using to access system services,
4-7
- request routing
 - across the cluster, 44-3
 - across the network, 30-46 to 30-47
 - by handle, 32-4
 - by specification, 32-4
 - defining for SRPs, 32-3 to 32-5
 - local exchange routing, 30-49 to
30-50
 - on SRPs, 30-49
- request routing across the cluster,
44-3
- request routing table, 30-45, 30-53
- request routing table, extending,
33-6
- Request, 30-2, 30-12, 30-53, 30-54,
32-11
- Request.sys, 33-7, 33-16
- RequestDirect, 30-15, 30-55, 32-11
- RequestRemote, 32-4, 32-18
- requests
 - global, 33-14
 - guidelines for defining, 33-14
 - local, 33-14
 - routing by handle, 30-34 to 30-41
 - routing by specification, 30-41 to
30-44
 - system, 33-14, 33-17 to 33-20
- RequestTemplate.txt, 33-14, 33-15,
33-16
- request-based operations, 4-7
- request-based system services,
28-2
- ReserveBusAddress, 42-8
- ReservePartitionMemory, 36-22
- reserving system-common numbers,
34-10
- ResetCommLine, 16-3, 16-5
- ResetDeviceHandler, 40-19, 40-30
- ResetFrame, 10-20
- ResetIbusHandler, 11-55, 11-59
- ResetMemoryLL, 6-8, 24-10, 24-9,
24-14
- ResetTimerInt, 37-7, 37-8, 40-31
- ResetTrapHandler, 40-28, 40-31
- ResetVideo, 10-9, 10-21
- ResetVideoGraphics, 10-21
- ResetXBusMIsr, 40-31, 41-10
- ResizeIOMap, 24-15
- ResizeSegment, 24-11, 24-17
- resizing segments, 24-12
- resource descriptor table, 26-4
- resource descriptor, 26-4
- resource handles
 - defined, 30-34
 - interpreting the bits in, 30-35 to
30-41
- resource ID, 26-4
- resource management, xli
- resource sharing, 18-4, 30-34
- resource type codes, 26-4
- resource work area, 26-4
- resources, defined, 26-2
- resources, system, 33-1
- Respond, 30-3, 30-14, 30-53, 30-54
- response exchange, 30-18
- restarting instructions, 40-26
- retrieving error message text,
26-21
- retrofitting overlay programs, 38-2

- return overlay descriptors, 38-7
 - ReuseAlias, 24-16
 - ReuseAliasLarge, 24-16
 - RgParam, 6-4, 6-6, 6-10
 - RgParamInit, 6-8, 6-10
 - RgParamSetSimple, 6-10
 - RIH (*See* raw interrupt handlers)
 - RkvsVersion, 26-39
 - RMOS (*See* running real mode applications)
 - ROD (*See* return overlay descriptors)
 - roll call, 44-2
 - routing by handle, 30-34 to 30-41
 - routing by specification, 30-41 to 30-44
 - routing code
 - values for expanding specifications, 30-44
 - values for routing requests, 30-44
 - routing request-based services, 34-5
 - RSAM (*See* record sequential access method)
 - RSAM buffer, 22-2
 - RsrcCopyFromRsrcSet, 26-11
 - RsrcEndSetAccess, 26-10
 - RsrcGetCountAllSetType, 26-12
 - RsrcGetCountSetType, 26-12
 - RsrcInitSession, 26-8
 - RsrcInitSetAccess, 26-11 to 26-12
 - RsrcSessionEnd, 26-8
 - RsrcSessionInit, 26-7
 - RSWA (*See* record sequential work area)
 - RS-232-C, 2-9, 8-8, 16-2
 - RS-422, 2-6, 2-9
 - RS-485, 2-6, 2-9
 - RTC (*See* realtime clock)
 - run files, 5-1, 5-3, 6-5, 24-6
 - run queues, 30-53, 31-7
 - running applications larger than available memory, 38-1
 - running real mode applications, 1-6
 - running state, 29-4
 - runtime dynamic linking, 39-20 to 39-21
 - RWA (*See* resource work area)
- ## S
- SA (*See* segment address)
 - SAM (*See* sequential access method)
 - SAMGen, 8-2
 - SbPrint, 26-17
 - SbPrint, 26-34
 - ScanToGoodRsRecord, 22-3
 - scheduler, 29-3
 - scheduling techniques
 - event-driven priority ordered, 29-2
 - time-based, 29-2
 - scratch volume, 12-6
 - screen attributes, 10-2
 - ScrollFrame, 10-20
 - SCSI (target) ID, 18-5, 18-7
 - SCSI access modes
 - exclusive mode (mx), 18-10
 - peek mode (mp), 18-10
 - shared mode (ms), 18-10
 - SCSI bus, 18-2
 - SCSI command structure
 - information transfer phases, 18-14 to 18-15
 - order of commands, 18-15
 - types of commands, 18-14
 - using to control devices, 18-16
 - SCSI device, 18-7

- SCSI error conditions, 18-16
- SCSI Manager
 - SCSI bus, 18-2
 - as SCSI initiator, 18-20
 - configuring a SCSI Manager, 18-7
 - electrical layer, 18-1
 - logical session layer, 18-2
 - overview, 18-4
 - physical layer, 18-1
 - relationship to devices, user programs, 18-5
 - relationship to the file system, 18-12 to 18-13
 - target mode, 18-20
 - transport protocol layer, 18-1
- SCSI manager target mode, 18-20 to 18-21
- SCSI passwords, 18-11
- SCSI path, 18-5 to 18-6
- SCSI peripheral device, 18-1
- SCSI sense data, 18-17 to 18-20
- SCSI target mode commands, 18-21
- SCSI (Small Computer Systems Interface), 18-1 to 18-29
 - ScsiCdbDataIn, 18-16, 18-27
 - ScsiCdbDataInAsync, 18-27
 - ScsiCdbDataOut, 18-16, 18-27
 - ScsiCdbDataOutAsync, 18-27
 - ScsiClosePath, 18-26, 18-9
 - ScsiManagerNameQuery, 18-26
 - ScsiOpenPath, 18-5, 18-26
 - ScsiQueryInfo, 18-26
 - ScsiQueryPathParameters, 18-9, 18-26
 - ScsiRequestSense, 18-27
 - ScsiReset, 18-27
 - ScsiSetPathParameters, 18-9, 18-26
 - ScsiTargetCdbCheck, 18-28
 - ScsiTargetCdbWait, 18-28
 - ScsiTargetDataReceive, 18-22, 18-28
 - ScsiTargetDataReceiveAsync, 18-28
 - ScsiTargetDataTransmit, 18-22, 18-28
 - ScsiTargetDataTransmitAsync, 18-28
 - ScsiTargetOperationsAbort, 18-29
 - ScsiWaitCdbAsync, 18-27
 - ScsiWaitTargetDataAsync, 18-29
- secondary task, 36-18, 38-15
- segment address, 4-12, 24-2, 24-3
- segment attributes, 39-10
- segment base address, 4-12
- segment descriptors, 4-12, 24-4
- segment elements, 5-3
- segment not-present, 40-26
- segment offset, 24-2
- segment, expand down, 24-1
- segment, expand up, 24-1
- segment, huge, 24-1
- segmentation models, 5-3
- segmented addressing, 24-3
- segments, 24-2
 - address, 4-12, 24-2, 24-3
 - code, 5-3, 24-4, 24-6
 - data, 24-4
 - defined, 4-10, 24-3
 - dynamic data, 24-5, 24-7
 - dynamic link library, 39-19
 - global, 39-9
 - huge, 24-4
 - instance, 39-9
 - limit, 24-4
 - nonshared, 39-11 (*See also* global segments)
 - offset, 24-3 to 24-4

- shared, 39-12 (*See also* instance segments)
- special use of dynamic link library
 - instance segments, 39-20
- static data, 5-3, 24-5, 24-6, 24-9
- types, 24-4
- selecting
 - file access methods, 20-6 to 20-7
 - keyboard translation tables, 11-23
 - nationalizable operations, 4-15
 - system services to write, 34-7
 - W-, X-, and Z-block buffers, 32-9
- selectively masking devices, 40-10
- self-exiting applications, 36-1
- self-loading applications, 36-1
- semaphores, xlii, 1-8, 39-13
 - defined, 31-1
 - handle, 31-1, 31-3
 - lock bit, 31-1, 31-5
 - owner, 31-3, 31-5
 - variable, 31-3, 31-17
- SemClear, 31-7, 31-8, 31-20, 31-21
- SemClearCritical, 31-9, 31-21
- SemClose, 31-5, 31-20
- SemLock, 31-7, 31-20
- SemLockCritical, 31-9, 31-21
- SemMuxWait, 31-13, 31-21
- SemNotify, 31-13
- SemOpen, 31-5, 31-20
- SemSet, 31-12, 31-21
- SemWait, 31-12, 31-21
- Send, 30-15, 30-55
- SendBreakC, 15-7
- sending a request, 32-8 to 32-9
- sending a response, 32-11
- sending and receiving messages,
 - 32-12 to 32-14
- sending data to the video and a second device, 26-17
- sending messages, 2-2, 30-12 to 30-15
- SeqAccessCheckpoint, 35-13
- SeqAccessClose, 35-12
- SeqAccessCtrl, 35-12
- SeqAccessDiscardBufferData,
 - 35-13
- SeqAccessModeQuery, 35-13
- SeqAccessModeSet, 35-13
- SeqAccessOpen, 35-12
- SeqAccessRead, 35-12
- SeqAccessRecoverBufferData,
 - 35-13
- SeqAccessStatus, 35-12
- SeqAccessWrite, 35-12
- sequential access byte streams, 8-9
- sequential access method
 - advantages, 8-1
 - customizing, 8-2
- Sequential Access Service
 - operations
 - SeqAccessCheckpoint, 35-13
 - SeqAccessClose, 35-12
 - SeqAccessCtrl, 35-12
 - SeqAccessDiscardBufferData,
 - 35-13
 - SeqAccessModeQuery, 35-13
 - SeqAccessModeSet, 35-13
 - SeqAccessOpen, 35-12
 - SeqAccessRead, 35-12
 - SeqAccessRecoverBufferData,
 - 35-13
 - SeqAccessStatus, 35-12
 - SeqAccessWrite, 35-12
- serial communications channels,
 - 8-8
- serial port interfaces, 16-1
- SerialNumberOldOsQuery, 27-12
- SerialNumberQuery, 27-13

- server workstation, 2-7, 2-8
- server, 2-6, 2-7, 30-1, 44-1, 44-2
- ServeRq, 30-53, 33-8, 33-10, 33-28, 34-4, 34-8
- service exchange, 2-2, 30-3, 30-18, 30-45, 30-53, 33-3, 30-53
- Set386TrapHandler, 40-27, 40-31
- SetAlphaColorDefault, 10-22
- SetBsLfa, 9-2, 9-5
- SetDaBufferMode, 23-5
- SetDateTime, 26-22, 26-37
- SetDateTimeMode, 26-37
- SetDefault386TrapHandler, 40-28, 40-31
- SetDefaultTrapHandler, 40-28, 40-31
- SetDeviceHandler, 40-19, 40-31
- SetDevParams, 13-5
- SetDirStatus, 12-46
- SetDiskGeometry, 13-5
- SetExitRunFile, 5-8
- SetFhLongevity, 12-22, 12-46
- SetField, 26-24, 26-25, 26-42
- SetFieldNumber, 26-24, 26-42
- SetFileStatus, 12-44
- SetIBusHandler, 11-55, 11-59
- SetImageMode, 9-2, 9-4
- SetImageModeC, 15-7
- SetIntHandler, 40-19, 40-28, 40-32
- SetIOOwner, 24-15
- SetKbdLed, 11-57
- SetKbdUnencoded, 11-12
- SetKbdUnencodedMode, 11-48, 11-58
- SetKeyboardOptions, 11-45, 11-58
- SetLdtRDs, 40-32
- SetLpIsr, 40-22, 40-32
- SetMsgRet, 5-8, 33-10
- SetNode, 12-45
- SetPartitionLock, 36-23
- SetPartitionName, 33-11, 33-28, 36-20
- SetPartitionSwapMode, 36-21
- SetPath, 12-10, 12-45
- SetPrefix, 12-10, 12-45
- SetRsLfa, 22-3
- SetScreenVidAttr, 10-9, 10-21
- SetSegmentAccess, 24-16
- SetStyleRam, 10-22
- SetStyleRamEntry, 10-22
- SetSwapDisable, 36-21
- SetSysInMode, 11-46, 11-48, 11-58
- SetTimerInt, 37-6, 37-8, 40-32
- setting
 - keyboard options, 11-45
 - reverse video or half-bright, 10-9
 - semaphores (See locking semaphores)
- setting up
 - a base RWA, 26-12
 - DLL search paths, 39-17
 - operating environments, 2-10
- SetTrapHandler, 40-27, 40-32
- SetUserFileEntry, 26-23, 26-7, 26-42
- SetVideoTimeout, 10-21
- SetXBusMIsr, 40-23, 40-32, 41-10
- SgFromSa, 24-17
- shared mode (ms), 18-10
- shared resource processor board
 - features, 2-9 to 2-10
- shared resource processor boards
 - cluster processor, 17-1
 - general processor, 2-7
 - storage processor, 2-7
 - terminal processor, 17-1
- shared resource processor, 2-4
- sharing code and data, 2-24

- sharing server run files, 12-29
- ShortDelay, 37-3, 37-8
- short-lived memory, 2-22, 5-6, 36-3, 36-7
- short-lived, 24-2
- ShrinkAreaLL, 24-9, 24-14
- ShrinkAreaSL, 24-9, 24-14
- ShrinkPartition, 24-15
- signaling and synchronizing
 - processes, 31-12
- signaling semaphores, equivalent CTOS functions, 31-15
- signaling semaphores, 31-3, 31-6
- SignOn password, 12-13
- SignOnLog, 26-43
- sizing
 - buffers, 23-2
 - programs, 5-3
- slot numbering, 32-2
- Small Computer Systems Interface (SCSI) management, 1-4, 1-6, 18-1 to 18-29
- small model, 24-6
- SnFromSr, 24-17
- software foundation, 1-2
- software interrupts, 40-25
- source file, 45-2
- source keyboard, 11-31
- SP (*See* storage processor)
- spanned records, 20-1
- special keys table, 27-6
- specification routing rules, 30-41
- specifying
 - default file passwords, 12-9
 - local file systems, 12-29
 - passwords, 12-8
 - screen coordinates and frame dimensions, 10-9
 - SCSI path parameters, 18-8
 - window sizes, 41-4
- Spooler operations
 - ConfigureSpooler, 35-14
 - SpoolerPassword, 35-14
- Spooler queue entries, 2-6
- SpoolerVersion, 26-39
- SrFromSn, 24-17
- SRP
 - name table, 32-4
 - request routing, 30-47 to 30-49
 - routing types, 30-49, 32-3 to 32-5
 - terminal management, 17-1
- stack, 39-4
- standard character set, 11-4
- standard file header, 20-6, 27-6
- standard operating system libraries, xxxviii
- standard record header, 23-1, 27-7
- standard record trailer, 20-6, 23-1
- starting a resource session, 26-7
- static data segment, 24-5, 24-6
- statically linked object modules, 39-4
- StaticsDesc, 38-6
- StatisticsVersion, 26-39
- status codes, 4-3, 4-7
- sticky keys, 11-45
- storage processor, 2-7
- storing
 - command form parameters, 6-5
 - parameter data, 6-3
- string
 - delimiters, 26-26
 - masks table, 27-7
 - offsets table, 27-7
 - table, 27-7
- string operations
 - comparing strings, 26-14
 - handling nationalized strings, 26-15
- strings table, 27-7

StringsEqual, 26-15, 26-31
structured file access methods,
 20-1
stub, 38-8
stubs array, 38-6
submit files, 11-54
 editing precautions, 11-50
 entering user data, 11-51
 escape sequence, 11-49 to 11-51
 playback, 11-47
 recording, 11-49
submit mode, 11-47
subparameters, 6-2 to 6-3, 6-4
SuperGen Series 5000, 41-1
suppressing duplication of volume
 control structures, 2-5
suspended state, 29-5
swap files, 36-14
SwapInContext, 36-21
swapper, 38-1
swapping requests, 33-19, 33-25
swapping, 2-19
SwapXBusEar, 41-10
synchronizing execution, 30-16
syntax errors in file specifications,
 12-40
Sys.keys, 11-33
SysGen, 43-1
SysInit.jcl, 33-8
system configuration block, 27-7
system configuring
 number of open system
 semaphores, 31-6
 W-, X-, and Z-blocks, 32-5
system configuration options (*See*
 system configuring)
system date/time format, 26-21
system date/time structure, 26-21,
 27-7
system events, 29-2

system initialization, 36-9
system input process, 11-46 to
 11-49
system keyboard file, 1-5, 11-18
 (*See also* NlsKbd.sys)
system memory organization, 2-14
 to 2-18
system performance, 3-14
system profile keyboard, 11-4,
 11-41
system request file, 33-7 (*See also*
 Request.sys)
system requests, 5-5, 33-28
system semaphores, 31-3, 31-5
system services
 built-in, 33-5
 comparing program model to
 client, 33-4
 comparing request-based to
 system-common, 34-3 to
 34-7
 components, 33-7
 converting to multi-instance
 service, 33-21
 defined, 33-1, 34-1
 deinstalling, 33-8, 33-26 to 33-27
 dynamically installable, 33-6
 execution model, 33-2
 guidelines for writing, 28-2, 33-8
 to 33-11
 initialization, 33-8
 multi-instance, 33-2
 overview of operation, 33-2 to
 33-5
 partition components, 33-11
 program model, 33-13
 request-based model, 34-1 to 34-2
 restrictions and requirements of
 operation, 33-14
system service exchange, 33-6

SystemCommonInstall, 34-9, 34-12
SystemCommonQuery, 34-12
system-common NLS table area,
 45-17
system-common numbers, 34-10,
 39-6
system-common procedures, 4-4
system-common services, 28-3
 deinstalling, 34-5, 34-8
 guidelines for writing, 34-8 to
 34-9
 model overview, 34-2

T

tape byte streams (*See* sequential
 access byte streams)
target ID, 18-5, 18-7
target keyboard, 11-31
target status, 18-16
task switch, 34-5
Telephone Service configuration
 block, 27-7
Telephone Service configuration file
 format, 27-7
Telephone status structure, 27-7
terminal output buffer, 27-7
terminal processor, 2-7
TerminatePartitionTasks, 36-16,
 36-23
terminating DLL clients, 39-15
terminating programs, 5-5 to 5-5,
 36-16
termination requests, 5-5, 33-18,
 36-16
text editing, 26-27
TextEdit, 26-27, 26-43
thread of execution, 34-2, 29-1
throughput, 18-5
time slicing, 29-2

timer pseudointerrupt blocks,
 27-7, 37-6
timer request blocks, 5-6, 27-8, 37-3
timing single events, 37-4
timing in 100 millisecond intervals,
 37-1
toggle, 11-4
token, 12-36
TP (*See* terminal processor)
TP boards, 17-1
TPIB (*See* timer pseudointerrupt
 block)
translating, 11-4
translation data block header, 27-8
translation table, 27-8
TransmitCommLineDma, 16-6
trap gate, 40-27
trap handlers, 40-27
traps, 40-4, 40-4
TRB (*See* timer request block)
TruncateDaFile, 23-4
TsVersion, 26-39
typematic key, 11-16
type-ahead buffer, 11-4, 11-9,
 11-16, 11-54
 discarding contents, 11-17 to
 11-17
 writing to, 11-17

U

UCB (*See* user control block)
ULCmpB, 26-31, 45-30
underrun, 40-11
unencoded mode, 11-4
unencoded plus, 11-12, 11-13
unencoded value, 11-4
UnlockInContext, 36-21
UnlockVideo, 10-23
UnlockVideoForModify, 10-23

- UnlockXbis, 41-10
 - UnmapBusAddress, 42-6, 42-8
 - UnzoomBox, 26-33
 - UpdateFpMountTable, 33-10
 - UpdateOverlayLru, 38-17
 - updating SRP master processor
 - name table, 33-10
 - user bit, 11-16
 - user control block, 12-34, 27-8
 - user number, 2-11, 2-13, 5-1, 11-16,
 - 28-3, 36-8, 36-14 (*See also* partition)
 - user structure, 11-16, 36-3, 36-5, 36-17
 - UserSysCommonLabel.asm, 34-9, 34-10
 - using byte streams, 8-3 to 8-4
 - using critical section semaphores, 31-9
 - using CTOS operations, 4-3 to 4-4
 - using default response exchanges, 30-18
 - using DMA buffers, 42-6
 - using ENLS operations, 4-15
 - using exchanges to transmit data, 30-4
 - using long-lived memory, 24-10
 - using medium model procedures in DLLs, 39-13
 - using multibyte strings, 11-40
 - using overlays 5-4 (*See also* virtual code management)
 - using parameter management, 6-2
 - using passwords for system access, 12-13
 - using path handles, 18-9
 - using PrintFile operations, 26-16 to 26-17
 - using requests for asynchronous I/O, 30-30
 - using Screen Setup to specify video characteristics, 10-10
 - using SCSI target check or wait operations, 18-23
 - using semaphores, 31-2
 - using short-lived memory, 24-11
 - using system-common procedures in a client, 34-11
 - using termination requests, 33-19
 - using the current screen setup, 10-3
 - using the Module Definition Utility, 39-11
 - using the SCSI manager, 18-23
 - using the stack, 39-14
 - using timer management operations, 37-2
 - using unique workstation hardware IDs, 26-27
 - using VAM
 - for advanced text processing, 10-11
 - for forms-oriented interaction, 10-10
 - using X-bus operations to access module memory, 41-3
 - utility operations, functions of, 26-1
- ## V
- V.35, 2-9, 16-2
 - VacatePartition, 36-16, 36-23
 - validating
 - file specifications, 12-38 to 12-40
 - protection levels, 12-16 to 12-17
 - VAM (*See* Video Access Method)

- variable length parameter block
 - 6-3, 6-4, 24-10, 27-8
 - constructing, 6-8
 - example, 6-6
 - initializing memory, 6-8
 - structure, 6-9
- variable partition memory
 - management, 2-12, 2-14 to 2-16, 3-3, 3-4, 4-13
- variable partition operating
 - systems, 36-1, 36-9, 36-11, 38-1, 38-8
 - multiple CPUs, 42-3
 - single CPU, 42-3
- variable partitions, 1-7, 2-13, 24-10, 36-7
- variable-length records, 20-1
- varying cache buffer size, 25-10
- VCB (*See* video control block)
- Video Access Method, 10-8
- VDM (*See* video display management)
- verify mode (*See* mode 3 DMA)
- VHB (*See* volume home block)
- video access method, 10-3
- video byte streams, 8-9
 - filtering to a file, 8-4
 - redirecting to a file, 5-4
- video byte streams special
 - characters, 10-5
- video control block, 10-17, 27-8
- video display management, 10-3, 10-8
- video frame 0, 26-16
- virtual 8086 mode, 1-1, 1-6, 3-13
- virtual circuit connection (*See* SCSI Path)
- virtual code management, 2-22, 24-7, 38-1
 - and the Linker, 38-4, 38-14
 - and virtual memory operating systems, 38-1
 - call/return conventions, 38-13
 - data structures, 38-4 to 38-7
 - model overview, 38-3 to 38-3
 - protected mode operation, 38-8 to 38-9, 38-15
 - real mode operation, 38-9 to 38-13, 38-14, 38-15
 - retrofitting overlay programs, 38-2
 - stack frame, 38-10
 - tracing the stack, 38-11 to 38-13
- virtual code management data
 - structures
 - overlay descriptors, 38-6
 - overlay zone header, 38-6
 - ProcInfoNonres, 38-7
 - ProcInfoRes, 38-6
 - return overlay descriptors, 38-7
 - StaticsDesc, 38-6
 - stubs array, 38-6
- virtual memory management,
 - xxxix, xl, 2-13, 2-16, 3-3, 4-13
- virtual memory operating systems,
 - 36-1, 36-9, 38-1, 40-28, 42-1
- virtual memory, xxxvii, 3-3
- VLPB (*See* variable length parameter block)
- Voice control structure, 27-8
- Voice file header, 27-8
- Voice file record, 27-8
- Voice Processor control block, 27-8

Voice/Data operations

- AsGetVolume, 35-14
- TsConnect, 35-14
- TsDataChangeParams, 35-14
- TsDataCheckpoint, 35-14
- TsDataCloseLine, 35-14
- TsDataOpenLine, 35-14
- TsDataRead, 35-15
- TsDataRetrieveParams, 35-15
- TsDataUnacceptCall, 35-15
- TsDataWrite, 35-15
- TsDeinstall, 35-15
- TsDial, 35-15
- TsDoFunction, 35-15
- TsGetStatus, 35-15
- TsHold, 35-15
- TsOffHook, 35-15
- TsOnHook, 35-16
- TsQueryConfigParams, 35-16
- TsReadTouchTone, 35-16
- TsRing, 35-16
- TsSetConfigParams, 35-16
- TsVoiceConnect, 35-16
- TsVoicePlaybackFromFile, 35-16
- TsVoiceRecordToFile, 35-16
- TsVoiceStop, 35-16

volatile memory (See memory addresses)

volume control structures, 13-4

- allocation bit map, 12-32
- bad sector file, 12-32
- device control block, 12-35
- directories, 12-33
- disk extent, 12-6, 12-32
- extension file header block, 12-32
- file area block, 12-34
- file control block, 12-34
- file header blocks, 12-6, 12-30, 12-32

I/O block, 12-34

- master file directory, 12-6, 12-33
- system directory, 12-33
- user control block, 12-34
- volume home block, 12-6, 12-30, 12-34

volume control structures,
duplication of, 2-5

volume encryption, 12-18

volume home block, 12-6, 12-11,
12-23, 12-34, 27-8

volume home block, accessing, 13-4

volume name, 13-2

volume passwords, 12-8, 12-12

W

wait exchange table, 32-7

wait flags table, 32-7

Wait, 30-16, 30-53, 30-54

waiting for several semaphores to
clear, 31-13

waiting state, 29-5

WaitLong, 30-54

wait-satisfied flags table, 32-7

wild card operations, 12-41

WildCardClose, 12-44

WildCardInit, 12-41, 12-44

WildCardMatch, 26-15, 26-32

WildCardNext, 12-41, 12-44

working set, 3-3, 3-9

workstation agent, 30-33, 30-34,
30-51

workstation operating system
features, 2-8

workstation video capabilities,
10-11 to 10-16

workstations
 server, 2-7
 local file system, 2-7
 diskless cluster workstations, 2-7
 EISA/ISA-bus, 13-4
 SuperGen Series 5000, 13-4
writable segments, 11-26
Write, 12-43
WriteAsync, 12-49, 30-30
WriteBsRecord, 8-15
WriteByte, 8-15
WriteByteStreamParameterC,
 15-7, 16-3
WriteCommLineStatus, 16-4, 16-5,
 40-32
WriteDaFragment, 23-2, 23-5
WriteDaRecord, 23-4
WriteHardId, 26-27, 26-43
WriteIBusDevice, 11-55, 11-59
WriteIBusEvent, 11-55, 11-59
WriteKbdBuffer, 11-17, 11-58
WriteLog, 26-28, 26-43
WriteRsRecord, 22-3
WriteStatusC, 15-7, 16-4
write-back, 25-2
write-behind mode, 23-3
write-through mode, 23-3
write-through, 25-2
writing a CTOS call, 4-3 to 4-4
writing device-independent
 programs, 7-3
writing language statements, 4-3
writing pages to backing store,
 3-11
writing records to the system log
 file, 26-28
writing to the type-ahead buffer,
 11-17
writing video programs for different
 workstation models, 10-16

W-block, 32-2, 32-5, 32-6

X

X.21, 2-9
X.25 byte streams, 8-9
X.25, 16-2
XBIF, 41-7
XbifVersion, 26-39
XBIS (*See* X-Bus initialization
 structure)
XC002Version, 26-39
X-Bus
 assigning I/O addresses to X-Bus
 modules, 41-1
 DMA, 41-6
 initialization structure, 41-6
 interrupts, 41-8
 management, 41-1 to 41-8
X-Bus initialization structure, 41-6
X-Bus modules
 accessing in protected mode, 41-5
 accessing in real mode, 41-5
 accessing module memory, 41-3
 communicating with the processor
 module, 41-6
 memory usage classes, 41-3
X-Bus interrupt handlers, 40-23
X-Bus+, 41-1

Y

Y-block, 32-2, 32-5, 32-6

Z

Z-block, 32-2, 32-5, 32-6
ZoomBox, 26-16, 26-33
ZPrint, 26-17, 26-34
[VID]0, 26-16



Help Us To Help You

Publication Title

Form Number

Unisys Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information. Please check type of suggestion:

☐ Addition

☐ Deletion

☐ Revision

☐ Error

Comments:

Name

Telephone number
()

Title

Company

Address

City

State

Zip code

Cut along dotted line ✂

Tape

Please Do Not Staple

Tape

Fold Here



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 817 DETROIT, MI

POSTAGE WILL BE PAID BY ADDRESSEE

UNISYS CORPORATION
PRODUCT INFORMATION
MS 18-007
2700 NORTH FIRST STREET
SAN JOSE, CA 95134-2028





43601186-000